Universidad de Los Andes

http://cesimo.ing.ula.ve

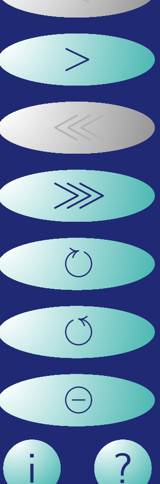
Unidad 5: Aplicaciones de la lógica computacional

Jacinto Dávila

mailto:jacinto@ula.ve

Centro de Simulación y Modelos (CESIMO)

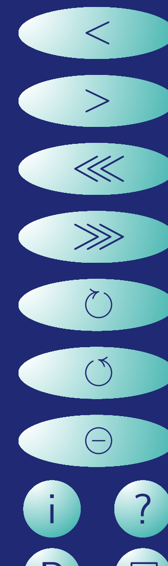
ULA LÓGICA



La unidad final del curso se refiere a algunas aplicaciones bien conocidas de la lógica computacional:

1. De nuevo los grafos (en PROLOG).
2. Algoritmos de búsqueda.
   - Algoritmo general.
   - Primero en profundidad.
   - Primero en anchura.
   - Búqueda informada.
3. Autómatas y máquinas de Turing (en PL).
4. Agentes y sistemas basados en conocimiento.

ULA LÓGICA



De nuevo los grafos en PROLOG

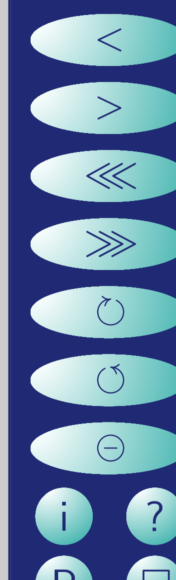
Otra formas de representar grafos (y por ende árboles) en PROLOG usan como referencia la matríz de incidencia:

```
[a - [b,c], b - [d,e], c - [b,f,g], d - [h,i], ...  , l - [k] ]

arco( X, Y, G) :- member(X-L,G), member( Y, L ).
```

ó así

```
incidencia(a, [b,c]).

incidencia(b, [d,e]).

incidencia(c, [b,f,g]).

incidencia(d, [h,i]).

incidencia(l, [k]).

arco(X,Y) :- incidencia( X, L), member( Y, L).
```

ULA LÓGICA



Arboles

Tanto estas como las formas anteriores para representar grafos se pueden usar para describir árboles en lógica. Sin embargo, dada la estructura particular de los árboles se usan todavía otras formas de representación, tales como la conocida:

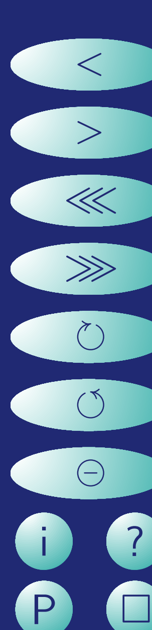
$$arbol(nodo, hijos)$$

donde nodo indica en nombre de la raíz del árbol y hijos es una lista con los hijos de este nodo. Esos hijos puede ser ellos mismos árboles o pueden ser nodos "hojas". Consideren el ejemplo anterior:

arbol(a, [arbol(b, [d,e]), arbol(c, [arbol(f, [g])])])

En la jerga de árboles se habla de árboles *and-or* (y-o) y de árboles de juegos. Estos árboles se caracterizan porque sus nodos se pueden clasificar en dos grupos (nodos *and* y nodos *or*, por ejemplo) y los nodos de un grupo sólo se conectan a nodos del otro grupo.

ULA LÓGICA



El árbol:

or(a, [and(b, [d, or(e, [i, and(j, [n, o])], k), f]), and(c, [or(g, [l, m]), h])])

podría usarse para representar el espacio de búsqueda vinculado al siguiente programa lógico:

$a \leftarrow b.$

$a \leftarrow c.$

$b \leftarrow d \wedge e \wedge f.$

$c \leftarrow g \wedge h.$

$e \leftarrow i.$

$e \leftarrow j.$

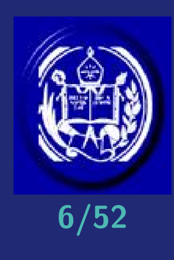
$e \leftarrow k.$

$j \leftarrow n \wedge o.$

$g \leftarrow l.$

$g \leftarrow m.$

ULA LÓGICA

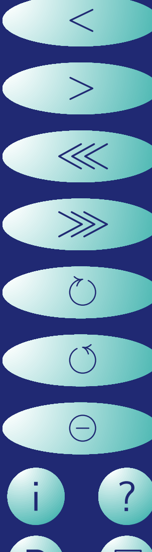


Así el problema de prueba de teoremas que hemos venido estudiando puede verse como un problema de búsqueda sobre un grafo (árbol).

¿Qué es lo que se busca?.

Actividad: Construya el árbol de resolución sobre ese programa y con la pregunta ?a. Compárelo con el árbol al comienzo de la lámina anterior.

ULA LÓGICA

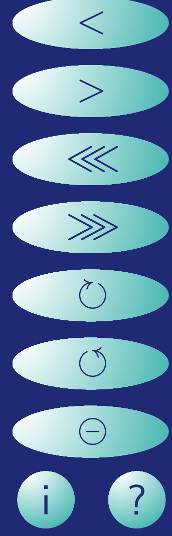


Algoritmos de búsqueda.

Las estrategias y los algoritmos de búsqueda son uno de los tópicos claves de la (vieja y buena) inteligencia artificial y de otras disciplinas. En esta sesión haremos una revisión general de las estrategias de búsqueda sobre grafos, revisaremos implementaciones en PROLOG y usaremos esas aplicaremos esos programas a un problema cotidiano de planificación.

Estrategias generales de búsqueda sobre grafos son:

ULA LÓGICA



1. **Búsqueda a fuerza bruta**:

   Búsqueda primero en profundidad.

   Búsqueda primero en anchura.

   "Profundizamiento progresivo (iterativo)".

2. **Búsqueda informada**:

   Búsqueda primero el mejor.

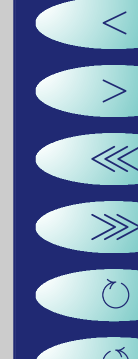
   Algoritmo A* (A estrella)

3. **Búsqueda en situaciones de adversidad**:

   Algoritmo minimax

   Algoritmo minimax mejorado: poda **alfa-beta**.

ULA LÓGICA



Los programas PROLOG que usamos en esta unidad pertenecen a:

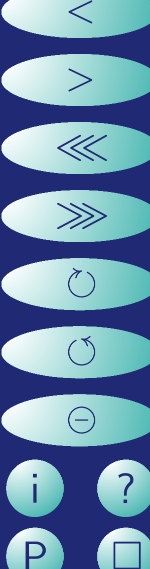
Shoham, Yoav. *Artificial Intelligence Techniques in PROLOG*. Morgan Kauffmann Publishers, Inc. 1994.

(ftp://unix.sri.com/pub/shoham).

Un clásico sobre búsqueda es: Pearl, Judea. *Heuristics*, Addison-Wesley, Reading. M.A. 1984.

Cualquier buen libro-texto sobre inteligencia artificial contiene una introducción al tópico de búsqueda. Recomiendo el texto de Norving and Russell de 1995.

ULA LÓGICA

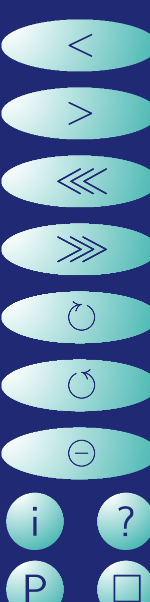


Técnicas de búsqueda sobre grafos

Dado un grafo G, un nodo $s \in G$ (el nodo de arranque o "*Start*") y un predicado $GoalPred$ (indicando cual es el estado objetivo o meta), el problema de búsqueda consiste en establecer si existe (y si existe mostrarlo) un camino desde s hasta un nodo t tal que $GoalPred(t)$ sea cierto.

La mayoría de los algoritmos de búsqueda emplean dos listas: la lista de nodos abiertos (OPEN) que contiene los nodos conocidos por el programa, pero aún no revisados y la lista de nodos cerrados (CLOSED) con aquellos ya revisados. La siguiente es la estructura básica de todos los algoritmos.

ULA LÓGICA



Procedimiento *búsqueda(G: grafo, Arranque: nodo, Meta: (Procedimiento unario))*

OPEN := { Arranque}, CLOSED := {}

encontrado := falso ;

mientras *OPEN no esté vacía y encontrado = falso* **haga**

mueve un nodo N de OPEN a CLOSED

si *Meta(N) es cierto* **entonces** *encontrado := cierto*

de lo contrario

encuentre todos los vecinos de N en el grafo que

no están ni en OPEN ni en CLOSED y agrégelos a OPEN

fin del si

fin del mientras

si *encontrado = cierto* **entonces** *devuelva N*

de lo contrario *falle*

ULA LÓGICA

Búsqueda primero en profundidad

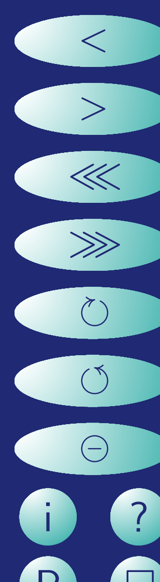
En este tipo de búsqueda, el algoritmo siempre selecciona de la lista OPEN, el último nodo que se le agregó. Se puede ver como un intento por extender un camino a lo largo del grafo hasta que ya no se le pueda extender ó hasta que se encuentre la meta. Si no se le puede extender más a cierto camino, la búsqueda "salta atrás" hacia el último punto en ese camino donde se dejó un nodo sin explorar (esto es *chronological backtracking*) y por allí continua extendiéndolo.

Observe que esto es exáctamente lo que hace el interpretador PROLOG.

Un código PROLOG para hacer búsqueda primero en profundidad se anexa a continuación y en el archivo **bbp**.

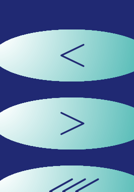
(Noten que hay más de una forma de implementar el algoritmo en PROLOG).

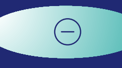
ULA LÓGICA



```
% Busqueda primero en profundidad.
% Argumentos: Start -> Nodo de partida.
%             GoalPred -> predicado con el objetivo.
%             Sol <- El camino desde el nodo de partida
%                    hasta el nodo indicado por GoalPred.
% Nota: Para cada aplicacion deben ser definidos los
%       predicados arc(X,Y) y el predicado indicado por GoalPred.
%  Depth-first search, without side effects

depth_first_search(Start, GoalPred, Sol) :-
    dfs([[Start]], [ ], GoalPred, Sol).
%  The first argument to dfs is the OPEN stack,
%  the second argument is the CLOSED list
dfs([[Node | Path]  |    _  ],  _  , GoalPred, [Node | Path]) :-
    Goal =.. [GoalPred,Node],
    Goal.
```

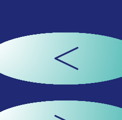
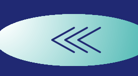
ULA, LÓGICA





```
dfs([[Node | Path] | MoreOPEN], CLOSED, GoalPred, Sol) :-
    % find the  new neighbors of the first OPEN node
    %and add the current path to each of them:
    findall(
        [Next,Node | Path],
        (
          arc(Node, Next),
          not(member([Next |  _  ], [[Node | Path]  | MoreOPEN])),
          not(member(Next, CLOSED))
        ),
        NewPaths
          ),
    % place the new paths on top of the stack:
    append(NewPaths, MoreOPEN, NewOPEN),
    dfs(NewOPEN, [Node | CLOSED], GoalPred, Sol).
```

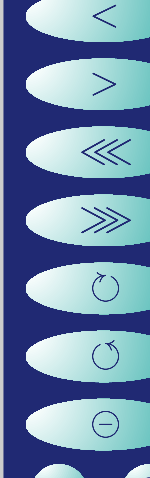
ULA, LÓGICA

```
% Este es otro programa, que hace lo mismo.
% Depth-first search, with side effects
depth_first_search_alt(Start, GoalPred, Sol) :-
    retractall(marked( _  )),
    asserta(marked(Start)),
    dfs_alt(Start, GoalPred, Sol), !,
    retractall(marked( _  )).
depth_first_search_alt( _  , _  , _  ) :-
    retractall(marked( _  )),
    fail.

dfs_alt(Node, GoalPred, [Node]) :-
    Goal =.. [GoalPred,Node],
    Goal.
dfs_alt(Node, Goal, [Node | Path]):-
    arc(Node, Next),
    unmarked(Next),
    dfs_alt(Next, Goal, Path).

unmarked(Node):-
    not(marked(Node)),
    assert(marked(Node)).
```


ULA LÓGICA

```
% --------------- Predicados Auxiliares

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

member( X, [X|_] ).
member( X, [_|Y] ) :- member( X, Y ).

% -------Un muy sencillo grafo como ejemplo.

arc( a, b).
arc( a, c).
arc( b, a).
arc( c, b).
arc( c, d).
meta(d).

% para arrancar el programa: ?depth_first_search(a, meta, Sol)
```



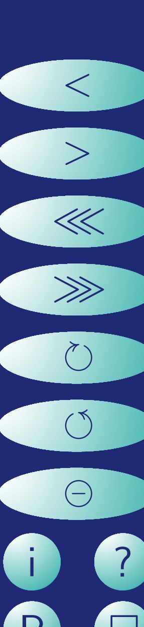
ULA LÓGICA

Búsqueda primero en anchura

Aunque la conducta del "explorador" es muy diferente y los requerimientos de recursos también, la búsqueda primero en anchura se obtiene como una simple variante del procedimiento de búsqueda primero en profundidad. Mientras en este último se selecciona el último nodo conocido por el programa, en búsqueda primero en anchura se selecciona el primer nodo conocido. Es como si, en lugar de manejar a la lista OPEN como una pila (LIFO), la manejaramos como una cola (FIFO)[a]. El algoritmo parece estar explorando el grafo "por capas" (incluso si el grafo no es un árbol). Como si un algoritmo tratara de explorar todos los caminos a partir del arranque en paralelo, pero corriendo en forma secuencial.

Los algoritmos de búsqueda primero en anchura son exigentes en memoria son inmunes a los grafos con caminos infinitos (es decir, pueden encontrar metas en grafos así) y también garantizan que encontrarán el camino más corto hasta el nodo meta.

[a] ¿Sabe Ud. que significa LIFO y FIFO?



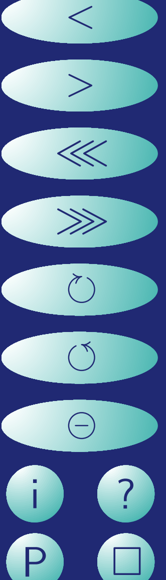
ULA LÓGICA

Observe que ni este programa (primero en anchura), ni el anterior (primero en profundidad), consiguen varios caminos a la misma meta. Se limitan a obtener algún camino a cada meta dada.

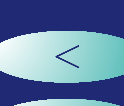
Un par de programas PROLOG para hacer búsqueda primero en anchura se encuentran a continuacion y en el archivo anexo **bpa**.

```
% Busqueda primero en anchura.
%Inefficient breadth-first search
%(almost identical to the depth-first search program)
breadth_first_search_slow(Start, GoalPred, Sol) :-
    bfs_slow([[Start]], [ ], GoalPred, Sol).
```

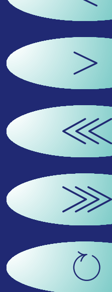
ULA LÓGICA



```
bfs_slow([[Node | Path]  |   _  ], _  , GoalPred, [Node | Path]) :-
    Goal =.. [GoalPred, Node],
    Goal.
bfs_slow([[Node | Path]  |  MoreOPEN], CLOSED, GoalPred, Sol) :-
    findall(
        [Next,Node | Path],
        (
          arc(Node, Next),
          not(lmember([Next |  _  ], [[Node | Path]  | MoreOPEN])),
          not(lmember(Next, CLOSED))
        ),
        NewPaths
          ),
    %place the new paths at the  bottom of the queue:
    append(MoreOPEN, NewPaths, NewOPEN),
    bfs_slow(NewOPEN, [Node | CLOSED], GoalPred, Sol).
```



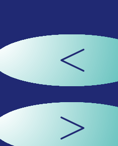
```
% More efficient breadth-first search
breadth_first_search(Start, GoalPred, Sol) :-
    bfs([[Start]  |  Qtail], Qtail, [ ], GoalPred, Sol).

%if the queue is empty, fail
bfs(OPEN,Qtail, _  , _  , _  ) :-
    OPEN==Qtail, !,
    fail.
%otherwise, as in the previous implementation:
bfs([[Node | Path]  |   _  ],  _  ,  _  , GoalPred, [Node | Path]) :-
    Goal =.. [GoalPred, Node],
    Goal.
```

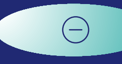


ULA LÓGICA

```
bfs([[Node | Path]  |  MoreOPEN], Qtail, CLOSED, GoalPred, Sol) :-
    findall(
        [Next,Node | Path],
        (
          arc(Node, Next),
          not dlmember([Next |  _  ],
                       [[Node | Path]  |  MoreOPEN], Qtail),
          not member(Next, CLOSED)
        ),
        NewPaths
           ),
    %and here is where the difference list pays off:
    append(NewPaths, NewQtail, Qtail),
    bfs(MoreOPEN, NewQtail, [Node | CLOSED], GoalPred, Sol).
```

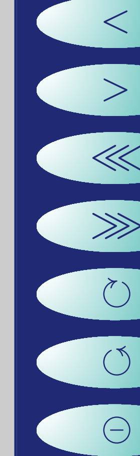
ULA LÓGICA





```
% ------------------------ Predicados Auxiliares
append( [], X, X ).
append( [X|Y], Z, [X|W] :- append( Y, Z, W ).

member( X, [X|_] ).
member( X, [_|Y] ) :- member( X, Y ).

dlmember( X, Head, Tail ) :- Head==Tail, !, fail.
dlmember( X, [X|_], _ ).
dlmember( X, [_|Y], Tail ) :- dlmember( X, Y, Tail ).

% ------------------ Usen el mismo grafo como ejemplo.
% para arrancar: ?breadth_first_search(a, meta, Sol)
```

ULA LÓGICA



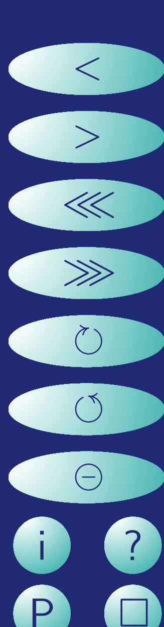
Actividad: Compare la eficiencia de los dos programas en el archivo bpa. Luego, investige que es una lista en diferencias en PROLOG (*difference-list*) y como se usan en uno de esos programas para lograr un notable aumento de eficiencia. *Pista*: imagine la siguiente forma de la relación agrega (o append) para agregar listas:

$$agrega_listas_en_dif((A, B), (B, C), (A, C))$$

donde A **y** B **son listas como** $[a, b, c|X]$ **literalmente.**

Pista: Trate representando esta lista como la tupla ([a,b,c—X], X).

ULA LÓGICA

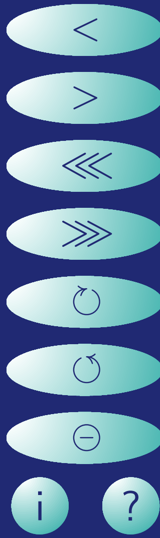


Planificación: Usando los algoritmos de búsqueda

Planificación automática es uno de las sub-áreas más interesantes de la Inteligencia Artificial. Básicamente se trata de hacer que un programa (en un computador) pueda generar planes para resolver ciertos problemas. La tradición dice que planificación es esencialmente un problema de búsqueda, aunque la opinión dominante es que esa búsqueda debe ser guiada con conocimiento específico del área del problema para que tenga éxito (es decir, la experiencia pesa).

Ignorando esa opinión (un poco), uno puede conseguir algunas aplicaciones sencillas de los algoritmos anteriores (basados en fuerza bruta de cómputo) a problemas de planificación “sencillos”.

Actividad: Tendremos un debate sobre planificación



ULA LÓGICA

Actividad: Considere la planificación de los cursos del postgrado. Se debe establecer quién dicta cuál curso y cuando, obedeciendo a ciertas restricciones y condiciones. La solución del problema (la meta) es una lista de esas especificaciones para cada curso que cumpla con las restricciones.

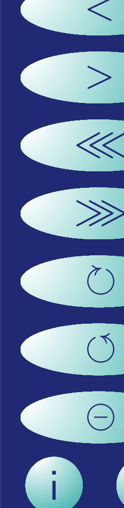
Usando curso(Nombre, Profesor, Semestre), la lista:

[curso(logica, jacinto, b2002), curso(tclf, ramon, a2003), curso(isoxo, judith, a2003),. . .]

es una posible solución (parcial).

Para obtenerla uno tendría que:

ULA LÓGICA



1. Definir la relación **arc** que usan los algoritmos, de manera que pueda generar una lista de cursos a partir de otra, por ejemplo:

   $arc(Lista, [curso(Nombre, Prof, Semestre)|Lista])$

   $\leftarrow (curso(Nombre, Prof, Semestre) \notin Lista \vee vacia(Lista))$

   $\wedge \neg curso_ya_programado(Nombre, Semestre, Lista)$

   $\wedge \neg profesor_ocupado(Prof, Semestre, Lista)$

   $\wedge puede_dictar(Profesor, Nombre)$

2. Definir la relación **meta** que usan los algoritmos, de manera que obtenga la planificación completa de los cursos, por ejemplo (Atención!.. hay formas más eficientes):

   $meta(Lista)$

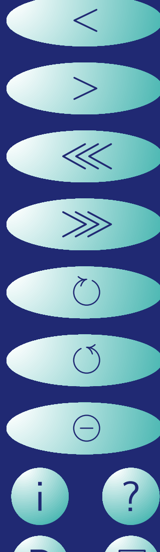
   $\leftarrow contiene_todos_los_cursos(Lista)$

   $\wedge ocupa_todos_los_profesores_pero_no_mucho(Lista)$

   $\wedge suficientes_materias_por_semestre(Lista)$

   $\wedge cada_cohorte_bien_atendida(Lista)$

ULA LÓGICA



Además de esas **especificaciones** operativas, tenemos que tomar en cuenta las **restricciones**:

Los cursos a considerar son, por ejemplo: *[logica, tclf, isoxo, cgraf, logdif, ayda, redes, sig, sd, ce, id]*.

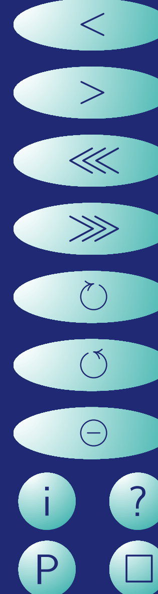
Los profesores (uno de los dominios de las variables) son, por ejemplo: *[jacinto, berti, jonas, domingo, wladimir, isabel, edgar, magdiel, tania, jose, judith]* y hay que decir quienes pueden dictar cuáles cursos.

Los semestres son, por ejemplo: [*a2002, b2002, a2003, b2004*]. Esto es importante. Limita el horizonte temporal. Las cohortes son, por ejemplo: vi, vii, viii.

Tenemos cursos específicos para la maestría y para la especialización. Ciertas materias (obligatorias) deben ser vistas por cada cohorte en el momento adecuado (como Uds. saben).

Como pueden imaginar, la complejidad es alta. Por eso se habla de que un planificador es una **aplicación inteligente.**

ULA LÓGICA

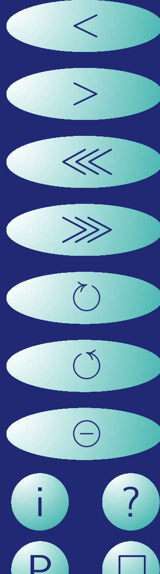


Profundizamiento iterativo-progresivo

La búsqueda primero en anchura tiene a su favor la búsqueda sistemática sobre todo el grafo. En su contra, normalmente es inútil, debido a los requerimientos de memoría, en problemas de mediana (realista) complejidad.

La búsqueda por **profundizamiento iterativo-progresivo** busca reunir lo mejor de esas dos técnicas anteriores (bpp y bpa), combinando la economía de memoria de la búsqueda primero en profundidad, bpp, con la revisión exhaustiva de todos los nodos que se hace con bpa.

ULA LÓGICA

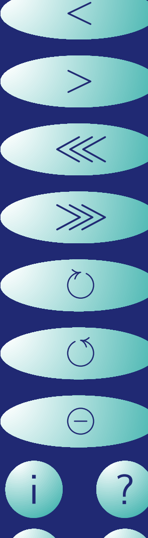


En profundizamiento iterativo-progresivo se usa un **contador de profundidad** para fijar un límite en la lóngitud de los caminos del grafo que se están explorando. Usando el mismo estilo de búsqueda primero en profundidad (el último nodo encontrado es el primero en ser revisado) se búsca exhaustivamente sobre todos esos caminos de longitud limitada.

Si no se encuentra una solución en ese "*horizonte*" limitado, se incrementa el contador de profundidad y se reinicia la búsqueda con la nueva longitud límite.

¿Qué tan difícil sería implementar esta estrategia aprovechando los programas PROLOG que hemos estado revisando en esta unidad?.

ULA LÓGICA



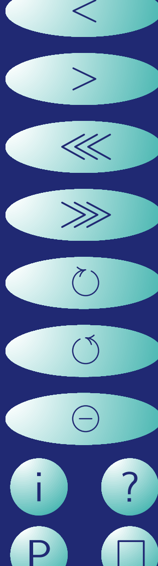
Búsqueda primero el mejor

Todos los métodos anteriores de búsqueda son métodos **ciegos** (o de fuerza bruta) puesto que el orden de búsqueda sobre el grafo se obtiene considerando su estructura únicamente, nunca su contenido (es decir, el tópico que almacenan).

Muchas veces se cuenta con información adicional (a la estructura del grafo) para "*guiar*" al explorador hacia el nodo meta más rápidamente.

Son todavía objeto de investigación las estrategias para codificar esa "*información adicional*" que puede ayudar al explorador. La estrategia tradicional es estimar la "bondad" de cada camino explorado y extender siempre al mejor (el de mayor bondad). De aquí el nombre de búsqueda primero el mejor.

ULA. LÓGICA



Esa medida de bondad es llamada el valor heurístico de cada camino y la función que se usa para computarlo es la funcion heurística.

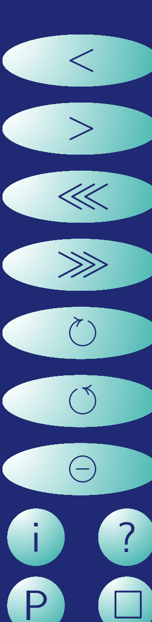
Algunas veces esta función heurística toma como única entrada el último nodo del camino.

Al tratar de implementar el algoritmo de búsqueda primero el mejor surgen dos complicaciones:

Primero, hay que modificar las listas OPEN y CLOSED para almacenar junto con cada camino su valor heurístico (pares $Valor - Camino$, en lugar de sólo Camino como antes).

La segunda complicación es: En las búsquedas ciegas uno puede ignorar múltiples caminos hacia el mismo nodo. En BPM esto no es posible, puesto que un nuevo camino al mismo nodo puede tener un mejor valor heurístico.

ULA LÓGICA



Suponga que el algoritmo acaba de encontrar un camino al nodo N.

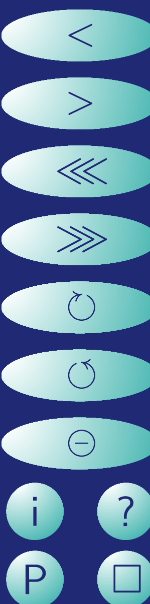
Si la lista OPEN contiene un camino a N con un peor valor heurístico, el nuevo camino debe reemplazar al viejo.

Si la lista CLOSED contiene al nodo N y el camino anterior a N tenía un peor valor (antes que el nodo fuese movido de OPEN a CLOSED), N debe ser removido de CLOSED y el nuevo camino puesto en OPEN (¿Por qué?).

Todo esto puede conducir a un procesamiento muy lento y pesado que podemos evitar si consideramos ciertas restricciones sobre la información almacenada.

El movimiento de nodos desde la lista CLOSED de regreso a OPEN puede producir una considerable sobrecarga. Ese movimiento no es necesario si el algoritmo es "*admisible-dondequiera*".

ULA LÓGICA



Definición lm5.1: Un algoritmo heurístico se dice "*admisible*" si encuentra, no sólo un camino a nodo meta, sino un camino de mínimo valor heurístico.

Definición lm5.2: Un algoritmo heurístico se dice "*admisible-dondequiera*" si, cuando quiera que extiende un camino desde un nodo x a su vecino, el camino hasta x ya es óptimo.

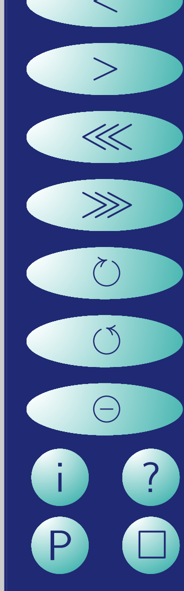
Admisible-dondequiera es una fuerta restricción sobre los algoritmos, pero no es poco común. De hecho, todo algoritmo cuya función heurística considera sólo el último nodo del camino (como dijimos antes) es *admisible-dondequiera*.

En estos algoritmos, como no es necesario devolver los nodos de CLOSED a OPEN, el código es más simple.

Actividad: Existe una definición de admisibilidad más estricta y gracias a la cual no es ni siquiera necesario revisar la lista OPEN. Se denomina "*admisibilidad-fuerte*". Investigue su definición. (Recuerde la bibliografía al principio de esta unidad).

Observe que todo lo dicho aplica para GRAFOS en general.

ULA LÓGICA



El algoritmo A*

$A*$(A estrella) es un caso especial de los algoritmos de búsqueda primero el mejor. Lo especial es su función heurística $f(n)$, la cual se compone de la forma siguiente:

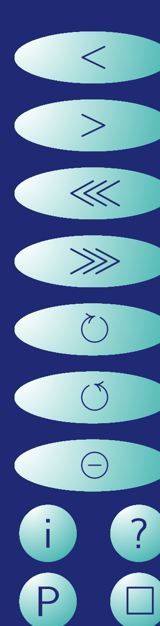
$$f(n) = g(n) + h(n)$$

donde:

$g(n)$ es la longitud del camino para alcanzar el nodo n.

$h(n)$ es una estimación de la longitud del camino desde nodo n hasta el nodo meta.

ULA LÓGICA

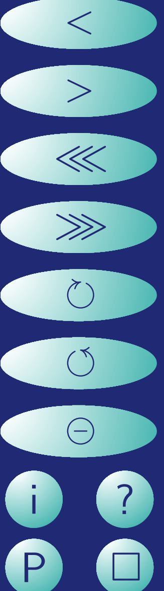


Un dato interesante para usar esta conocidísima estrategia de búsqueda es que si h es una función monótona, entonces A^* es "*admisible dondequiera*".

Definición lm5.3: h es monótona si satisface la desigualdad del triángulo. Es decir, si $arc(p, q)$, entonces $h(p) + 1 > h(q)$.

Existe un voluminoso substrato teórico describiendo las propiedades de todas estas estrategias de búsqueda. Este tópico es, probablemente, el mejor desarrollado en Inteligencia Artificial.

ULA LÓGICA



Autómatas y programación lógica

El autómata supervisor en PROLOG será descrito en una presentación anexa)

transicion(e0, t17, e1).

transicion(e1, t2, e2).

transicion(e2, t19, e3).

transicion(e3, t3, e4).

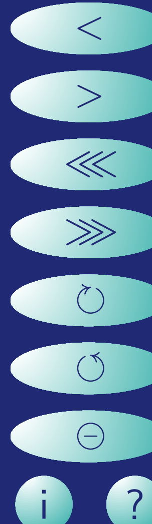
transicion(e4, t21, e5).

transicion(e5, t5, e6).

transicion(e6, t23, e7).

El predicado transición(X,Y,Z) dice que hay una transición Y, en el autómata, entre el estado X y el estado Z.

Para definir la dinámica de un sistema con un autómata es necesario definir TODAS las transiciones y, desde luego, todos los estados reciben un identificador.



ULA, LÓGICA

El autómata supervisor ¿no sería mejor así?

Si todo en calma entonces abra válvula_mezcla(X).

Si mezcla(X) y no punto_de_consigna entonces válvula_mezcla(X+ϵ).

Si mezcla(35%) entonces encienda_bomba(uno).

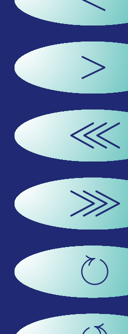
Si mezcla(55%) entonces encienda_bomba(dos).

Si mezcla(75%) y no punto_de_consigna entonces encienda_bomba(tres).

Si mezcla(95%) y no punto_de_consigna y tres_bombas_encendidas entonces abra válvula_químico.

Si mezcla(75%) y punto de consigna y válvula_químico_abierta entonces cierre válvula_químico.

ULA LÓGICA



La máquina de Turing

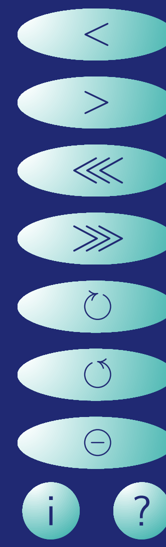
La máquina de Turing es un modelo ideal de un computador.

Esta compuesta, en su forma más usada, por una cinta infinita y un cabezal que se desplaza sobre esa cinta, en ambas direcciones, leyendo y escribiendo.

Quizás lo más impresionantes es que sea posible pensar en una máquina universal que puede simular a cualquier otra máquina, siempre que el programa de esa otra máquina se le suministre a la universal... grabado en la cinta.

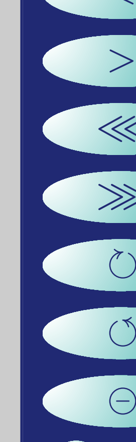
Eso haremos con la siguiente versión de la máquina universal escrita en PROLOG.

ULA LÓGICA



```
% mt.pl
% maquina de turing.
%

% predicado auxiliar para invocar la maquina..
m( [C|R] ) :- mt( restuna, cinta([], C, R), hacia(derecha) ).


% la maquina..
mt( Programa, Cinta, EdoActual ) :-
   paso( Programa, EdoActual, Cinta, NuevoEdo, NuevaCinta ),
   write( NuevaCinta ), nl,
   mt( Programa, NuevaCinta, NuevoEdo ).

mt( _, Cinta, stop ) :- write( stop ), nl.
```

ULA LÓGICA

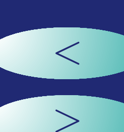


```
% para salir de un estado..  sin cambiar la cinta.
paso( P,
      hacia(Dir),                          % Estado Actual
      cinta(Izq, Cab, Der),                % Cinta Actual
      hacia(NDir),                         % Nuevo Estado
      NuevaCinta ) :-                      % Nueva Cinta
   programa( P, [hacia(Dir),
      buscando(Cab), pasa_a(NDir)] ),
   mueve_cinta( Dir, Izq, Cab, Der, NuevaCinta ).

% para salir de un estado..  cambiando la cinta.
paso( P,
      hacia(Dir),                          % Estado Actual
      cinta(Izq, Cab, Der),                % Cinta Actual
      hacia(NDir),                         % Nuevo Estado
      NuevaCinta ) :-                      % Nueva Cinta
   programa( P, [hacia(Dir),
      buscando(Cab), escribe(Simb), pasa_a(NDir)] ),
   mueve_cinta( Dir, Izq, Simb, Der, NuevaCinta ).
```

ULA LÓGICA

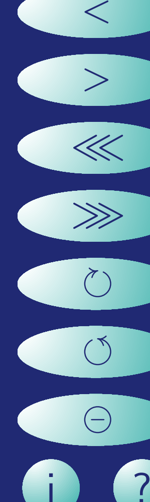
```
% para progresar sobre la cinta, en el mismo estado..
paso( P,
      hacia(Dir),                             % Estado Actual
      cinta(Izq, Cab, Der),                   % Cinta Actual
      hacia(NDir),                            % Nuevo Estado
      CintaFinal ) :-                         % Nueva Cinta
   ( programa( P, [hacia(Dir), buscando(Simb), escribe(_),
        pasa_a(NDir)] ) ;
     programa( P, [hacia(Dir), buscando(Simb), pasa_a(NDir)] )),
   not( Simb = Cab ),
   mueve_cinta( Dir, Izq, Cab, Der, NuevaCinta ),
   paso( P, hacia(Dir), NuevaCinta, hacia(NDir), CintaFinal ),
   write( NuevaCinta), nl.
```

ULA LÓGICA



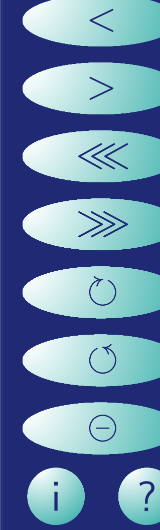
```
% para parar.
paso( P,
      hacia(Dir),                         % Estado Actual
      cinta(Izq, Cab, Der),               % Cinta Actual
      stop,                               % Nuevo Estado
      cinta(Izq, Simb, Der) ) :-          % Nueva Cinta
   ( programa( P, [hacia(Dir), buscando(Cab), escribe(Simb), stop] ) ;
     ( programa( P, [hacia(Dir), buscando(Cab), stop] ), Simb = Cab ) ).
```

ULA LÓGICA



```
% movimientos de la cinta
mueve_cinta( derecha, Izq, Cab, [NCab|NDer], cinta( NIzq, NCab,
NDer ) ) :-
   append( Izq, [Cab], NIzq ).

mueve_cinta( izquierda, Izq, Cab, Der, cinta( NIzq, NCab,
[Cab|Der] ) ) :-
   append( NIzq, [NCab], Izq ).
```

ULA LÓGICA



```
% programas para la maquina, descritos como en cierta notacion grafica.

% para la resta unaria..

programa( restuna, [hacia(derecha), buscando(num),
pasa_a(derecha)] ).

programa( restuna, [hacia(derecha),
buscando(1),pasa_a(izquierda)]).

programa( restuna,[hacia(izquierda), buscando(1), escribe(0),
pasa_a(derecha)] ).

programa( restuna, [hacia(izquierda), buscando(b), stop] ).

programa( restuna, [hacia(derecha), buscando(b), stop] ).
```

ULA LÓGICA

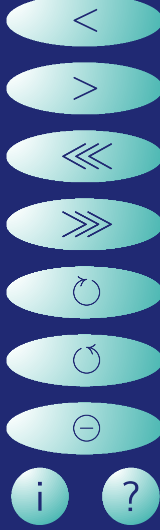
Agentes en programación lógica

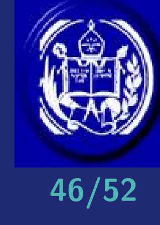
Les presento a GLORIA:

GLORIA′s cycle
$cycle(KB, Goals, T)$
$\leftarrow \; demo(KB, Goals, Goals', R)$
$\wedge \; R \; \leq n$
$\wedge \; act(KB, Goals', Goals'', T + R)$
$\wedge \; cycle(KB, Goals'', T + R + 1)$ [**GLOCYC**]

Table 1: El ciclo del agente en lógica

ULA LÓGICA

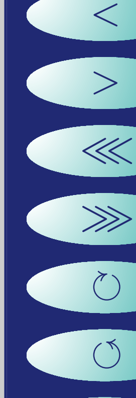


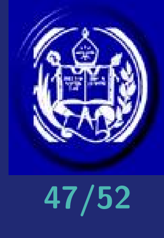


GLORIA′s act
$act(KB, Goals, Goals'', T_a)$
$\leftarrow\ Goals \equiv\ PreferredPlan\ \vee\ AltGoals$
$\wedge\ executables(PreferredPlan, T_a, TheseActions)$
$\wedge\ try(TheseActions, T_a, Feedback)$
$\wedge\ assimilate(Feedback, Goals, Goals')$
$\wedge\ use_order(Goals', Goals'', R_{use})$
$\wedge\ R_{use} \leq k$ [**GLOACT**]

Table 2: Qué significa que el agente actúe

ULA LÓGICA

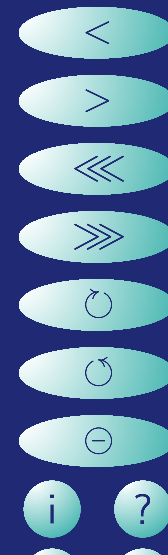




GLORIA′s executables
$executables(Intentions, T_a, NextActs)$ $\leftarrow\ \forall A, T(\ do(A,T)\ is_in\ Intentions$ $\wedge\ consistent((T = T_a) \wedge Intentions)$ $\leftrightarrow\ do(A, T_a)\ is_in\ NextActs\ \ )$ **[GLOEXE]**

Table 3: Cuáles intenciones son ejecutables

ULA LÓGICA

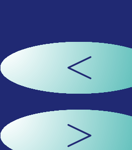


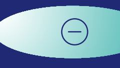
GLORIA′s assimilate

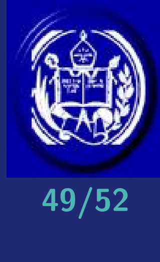
$$
\begin{aligned}
& assimilate(Inputs, InGoals, OutGoals) \\
& \quad \leftarrow \forall A, T, T'(\ action(A, T, succeed)\ is_in\ Inputs \\
& \qquad\qquad \wedge\ do(A, T')\ is_in\ InGoals \\
& \qquad\qquad \rightarrow\ do(A, T)\ is_in\ NGoal\ ) \\
& \quad \wedge\ \forall A, T, T'(\ action(A, T, fails)\ is_in\ Inputs \\
& \qquad\qquad \wedge\ do(A, T')\ is_in\ InGoals \\
& \qquad\qquad \rightarrow\ (\mathbf{false}\ \leftarrow\ do(A, T))\ is_in\ NGoal\ ) \\
& \quad \wedge\ \forall P, T(\ obs(P, T)\ is_in\ Inputs \\
& \qquad\qquad \rightarrow\ obs(P, T)\ is_in\ NGoal\ ) \\
& \quad \wedge\ \forall Atom(\ Atom\ is_in\ NGoal \\
& \qquad\qquad \rightarrow\ Atom\ is_in\ Inputs \\
& \quad \wedge\ OutGoals\ \equiv\ NGoal\ \wedge\ InGoals \qquad\qquad [\mathbf{GLOASSI}]
\end{aligned}
$$

Table 4: Asimilando la información que recibe

UNA LÓGICA







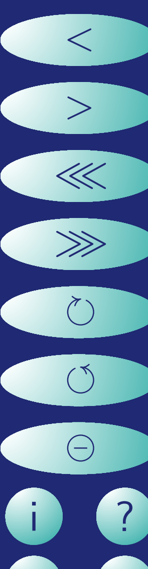
GLORIA's try	
$A\ is_in\ B \leftarrow B \equiv A \wedge Rest$	[**GLOISN**]
$try(Output, T, Feedback) \leftarrow$ *tested by the environment..*	[**TRY**]

Table 5: El punto de contacto con el ambiente

¿Cómo se leen estas fórmulas en español?

Actividad: Discuta que le falta a esta definición de agente.

ULA LÓGICA



Fin unidad 5

En esta unidad hemos hecho un recorrido superficial por programas de aplicación escritos en lógica. Estos pueden ofrecerse como una muestra de que la programación lógica puede usarse para crear programas reales y complejos.

La prueba real de la utilidad de la programación lógica debe obtenerse en escenarios realistas de desarrollo de software.

Sin embargo, la versatilidad del lenguaje debe ser ahora evidente cuando se considera que se le puede usar como lenguaje de especificación y como lenguaje de programación.

Nos interesa saber su opinión al respecto.

GRACIAS.

ULA LÓGICA