Tema 4: Resolución de problemas de espacios de estados

José A. Alonso Jiménez

Jose-Antonio.Alonso@cs.us.es
http://www.cs.us.es/∼jalonso

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Ejemplo de problema de EE: El 8–puzzle

- **Problema del 8–puzzle:**
  Para el 8–puzzle se usa un cajón cuadrado en el que hay situados 8 bloques cuadrados. El cuadrado restante está sin rellenar. Cada bloque tiene un número. Un bloque adyacente al hueco puede deslizarse hacia él. El juego consiste en transformar la posición inicial en la posición final mediante el deslizamiento de los bloques. En particular, consideramos el estado inicial y final siguientes:

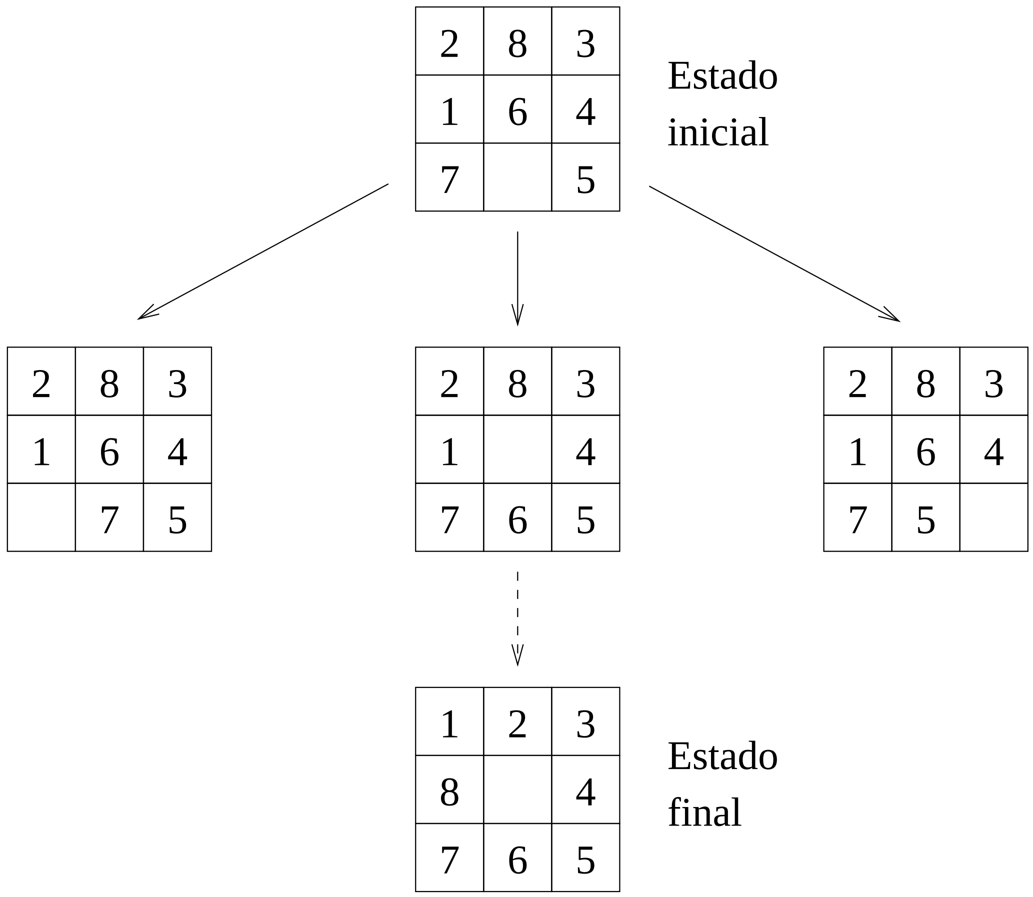
2	8	3
1	6	4
7		5

Estado inicial

1	2	3
8		4
7	6	5

Estado final

Ejemplo de problema de EE: El 8–puzzle



Ejemplo de problema de EE: El 8–puzzle

- **Solución del 8–puzzle:**

2	8	3
1	6	4
7		5

→

2	8	3
1		4
7	6	5

→

2		3
1	8	4
7	6	5

→

	2	3
1	8	4
7	6	5

→

1	2	3
	8	4
7	6	5

→

1	2	3
8		4
7	6	5

(arriba arriba izquierda abajo derecha)

Ejemplo de problema de EE: El 8–puzzle

- **Representación:**
  - **Estado inicial:** `[[2,8,3],[1,6,4],[7,h,5]]`
  - **Estado final:** `[[1,2,3],[8,h,4],[7,6,5]]`
  - **Operadores:**
    - **Mover el hueco a la izquierda**
    - **Mover el hueco arriba**
    - **Mover el hueco a la derecha**
    - **Mover el hueco abajo**
- **Número de estados = 9! = 362.880.**

Ejemplo de problema de EE: Problema del árbol

- **Árbol**

```
         a
       /   \
     b       c
    /  \    /  \
   d    e  f    g
 /     /  \  \
h     i    j   k
```

  - **Estado inicial:** `a`
  - **Estados finales:** `f` **y** `j`

Ejemplo de problema de EE: Problema del árbol

- **Representación** `arbol.pl`
  - `estado_inicial(?E)` **se verifica si** `E` **es el estado inicial.**

```
estado_inicial(a).
```

  - `estado_final(?E)` **se verifica si** `E` **es un estado final.**

```
estado_final(f).
estado_final(j).
```

  - `sucesor(+E1,?E2)` **se verifica si** `E2` **es un sucesor del estado** `E1`**.**

```
sucesor(a,b).
sucesor(a,c).
sucesor(b,d).
sucesor(b,e).
sucesor(c,f).
sucesor(c,g).
sucesor(d,h).
sucesor(e,i).
sucesor(e,j).
sucesor(f,k).
```

Búsqueda en profundidad sin ciclos

- `profundidad_sin_ciclos(?S)` **se verifica si** `S` **es una solución del problema mediante búsqueda en profundidad sin ciclos. Por ejemplo,**

```
?- [arbol].
?- profundidad_sin_ciclos(S).
S = [a, b, e, j]
?- trace(estado_final,+call), profundidad_sin_ciclos(S).
 T Call:  (9) estado_final(a)
 T Call: (10) estado_final(b)
 T Call: (11) estado_final(d)
 T Call: (12) estado_final(h)
 T Call: (11) estado_final(e)
 T Call: (12) estado_final(i)
 T Call: (12) estado_final(j)
S = [a, b, e, j]
?- nodebug.
```

Búsqueda en profundidad sin ciclos

```
profundidad_sin_ciclos(S) :-
   estado_inicial(E),
   profundidad_sin_ciclos(E,S).

profundidad_sin_ciclos(E,[E]) :-
   estado_final(E).
profundidad_sin_ciclos(E,[E|S1]) :-
   sucesor(E,E1),
   profundidad_sin_ciclos(E1,S1).
```

Búsqueda en profundidad sin ciclos: 4 reinas

- **Resolución del problema de las 4 reinas**
  - **Enunciado: Colocar 4 reinas en un tablero rectangular de dimensiones 4 por 4 de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.**
  - **Estados: listas de números que representa las ordenadas de las reinas colocadas. Por ejemplo, [3,1] representa que se ha colocado las reinas (1,1) y (2,3).**
  - **Soluciones:**

```
?- ['4-reinas','b-profundidad-sin-ciclos'].
Yes
?- profundidad_sin_ciclos(S).
S = [[], [2], [4, 2], [1, 4, 2], [3, 1, 4, 2]] ;
S = [[], [3], [1, 3], [4, 1, 3], [2, 4, 1, 3]] ;
No
```

- **Representación del problema de las 4 reinas (`4-reinas.pl`)**
  - `estado_inicial(?E)` **se verifica si** `E` **es el estado inicial.**

```
estado_inicial([]).
```

Búsqueda en profundidad sin ciclos: 4 reinas

- `estado_final(?E)` **se verifica si** `E` **es un estado final.**

```
estado_final(E) :-
   length(E,4).
```

- `sucesor(+E1,?E2)` **se verifica si** `E2` **es un sucesor del estado** `E1`**.**

```
sucesor(E,[Y|E]) :-
   member(Y,[1,2,3,4]),
   not(member(Y,E)),
   no_ataca(Y,E).
```

- `no_ataca(Y,E)` **se verifica si** `E=[Yn,...,Y1]`**, entonces la reina colocada en** `(n+1,Y)` **no ataca a las colocadas en** `(1,Y1)`**, ...,** `(n,Yn)`**.**

```
no_ataca(Y,E) :-
   no_ataca(Y,E,1).
no_ataca(_,[],_).
no_ataca(Y,[Y1|L],D) :-
   Y1-Y =\= D,
   Y-Y1 =\= D,
   D1 is D+1,
   no_ataca(Y,L,D1).
```

Búsqueda en profundidad con ciclos: Grafo

- **Grafo con ciclos**

```
         a
       /   \
     b       c
    /  \    /  \
   d    e  f    g
 //    /  \  \
h     i    j   k
```

  - **Estado inicial:** `a`
  - **Estados finales:** `f` **y** `j`
  - **Nota: el arco entre** `d` **y** `h` **es bidireccional**

Búsqueda en profundidad con ciclos: Grafo

- **Representación** `grafo.pl`
  - `estado_inicial(?E)` **se verifica si** `E` **es el estado inicial.**

```
estado_inicial(a).
```

  - `estado_final(?E)` **se verifica si** `E` **es un estado final.**

```
estado_final(f).
estado_final(j).
```

  - `sucesor(+E1,?E2)` **se verifica si** `E2` **es un sucesor del estado** `E1`**.**

```
sucesor(a,b).
sucesor(a,c).
sucesor(b,d).
sucesor(b,e).
sucesor(c,f).
sucesor(c,g).
sucesor(d,h).
sucesor(e,i).
sucesor(e,j).
sucesor(f,k).
sucesor(h,d).
```

Búsqueda en profundidad con ciclos: Grafo

- **Solución**

```
?- ['grafo','b-profundidad-sin-ciclos'].
?- trace(estado_final,+call), profundidad_sin_ciclos(S).
T Call:  ( 10) estado_final(a)
T Call:  ( 11) estado_final(b)
T Call:  ( 12) estado_final(d)
T Call:  ( 13) estado_final(h)
T Call:  ( 14) estado_final(d)
T Call:  ( 15) estado_final(h)
....

?- ['b-profundidad-con-ciclos'].
?- profundidad_con_ciclos(S).
S = [a, b, e, j] ;
S = [a, c, f] ;
No
```

Búsqueda en profundidad con ciclos: Grafo

```
?- trace(estado_final,+call), profundidad_con_ciclos(S).
 T Call: (10) estado_final(a)
 T Call: (11) estado_final(b)
 T Call: (12) estado_final(d)
 T Call: (13) estado_final(h)
 T Call: (12) estado_final(e)
 T Call: (13) estado_final(i)
 T Call: (13) estado_final(j)
S = [a, b, e, j] ;
 T Call: (11) estado_final(c)
 T Call: (12) estado_final(f)
S = [a, c, f] ;
 T Call: (13) estado_final(k)
 T Call: (12) estado_final(g)
No
```

Búsqueda en profundidad con ciclos

- **Procedimiento de búsqueda en profundidad con ciclos (`b-profundidad-con-ciclos.pl`)**
  - **Un *nodo* es una lista de estados $[E_n, \ldots, E_1]$ de forma que E_1 es el estado inicial y E_{i+1} es un sucesor de E_i.**
  - `profundidad_con_ciclos(?S)` **se verifica si** `S` **es una solución del problema mediante búsqueda en profundidad con ciclos.**

```
profundidad_con_ciclos(S) :-
   estado_inicial(E),
   profundidad_con_ciclos([E],S).

profundidad_con_ciclos([E|C],S) :-
   estado_final(E),
   reverse([E|C],S).
profundidad_con_ciclos([E|C],S) :-
   sucesor(E,E1),
   not(memberchk(E1,C)),
   profundidad_con_ciclos([E1,E|C],S).
```

Búsqueda en profundidad con ciclos: Problema de las jarras

- **Enunciado:**
  - **Se tienen dos jarras, una de 4 litros de capacidad y otra de 3.**
  - **Ninguna de ellas tiene marcas de medición.**
  - **Se tiene una bomba que permite llenar las jarras de agua.**
  - **Averiguar cómo se puede lograr tener exactamente 2 litros de agua en la jarra de 4 litros de capacidad.**
- **Representación:**
  - **Estado inicial:** `0-0`
  - **Estados finales:** `2-y`

Búsqueda en profundidad con ciclos: Problema de las jarras

- **Operadores:**
  *** Llenar la jarra de 4 litros con la bomba.**
  *** Llenar la jarra de 3 litros con la bomba.**
  *** Vaciar la jarra de 4 litros en el suelo.**
  *** Vaciar la jarra de 3 litros en el suelo.**
  *** Llenar la jarra de 4 litros con la jarra de 3 litros.**
  *** Llenar la jarra de 3 litros con la jarra de 4 litros.**
  *** Vaciar la jarra de 3 litros en la jarra de 4 litros.**
  *** Vaciar la jarra de 4 litros en la jarra de 3 litros.**

- **Solución**

```
?- ['jarras','b-profundidad-con-ciclos'].
?- profundidad_con_ciclos(S).
S = [0-0, 4-0, 4-3, 0-3, 3-0, 3-3, 4-2, 0-2, 2-0] ;
S = [0-0, 4-0, 4-3, 0-3, 3-0, 3-3, 4-2, 0-2, 2-0, 2-3]
Yes
?- findall(_S,profundidad_con_ciclos(_S),_L), length(_L,N).
N = 27
```

Búsqueda en profundidad con ciclos: Problema de las jarras

- **Representación (`jarras.pl`)**
  - `estado_inicial(?E)` **se verifica si** `E` **es el estado inicial.**

    ```
    estado_inicial(0-0).
    ```

  - `estado_final(?E)` **se verifica si** `E` **es un estado final.**

    ```
    estado_final(2-_).
    ```

  - `sucesor(+E1,?E2)` **se verifica si** `E2` **es un sucesor del estado** `E1`**.**

    ```
    sucesor(X-Y,4-Y)     :- X < 4.
    sucesor(X-Y,X-3)     :- Y < 3.
    sucesor(X-Y,0-Y)     :- X > 0.
    sucesor(X-Y,X-0)     :- Y > 0.
    sucesor(X1-Y1,4-Y2) :- X1 < 4, T is X1+Y1, T >= 4, Y2 is Y1-(4-X1).
    sucesor(X1-Y1,X2-3) :- Y1 < 3, T is X1+Y1, T >= 3, X2 is X1-(3-Y1).
    sucesor(X1-Y1,X2-0) :- Y1 > 0, X2 is X1+Y1, X2 < 4.
    sucesor(X1-Y1,0-Y2) :- X1 > 0, Y2 is X1+Y1, Y2 < 3.
    ```

Búsqueda en anchura: Problema del paseo

- **Problema del paseo**
  - **Enunciado: Una persona puede moverse en línea recta dando cada vez un paso hacia la derecha o hacia la izquierda. Podemos representarlo mediante su posición `X`. El valor inicial de `X` es 0. El problema consiste en llegar a la posición -3.**
  - `estado_inicial(?E)` **se verifica si** `E` **es el estado inicial.**

```
estado_inicial(0).
```

  - `estado_final(?E)` **se verifica si** `E` **es un estado final.**

```
estado_final(-3).
```

  - `sucesor(+E1,?E2)` **se verifica si** `E2` **es un sucesor del estado** `E1`**.**

```
sucesor(E1,E2) :-
   E2 is E1+1.
sucesor(E1,E2) :-
   E2 is E1-1.
```

Búsqueda en anchura: Problema del paseo

- **Solución:**
  - **Por búsqueda en profundidad con ciclos**

```
?- ['paseo','b-profundidad-con-ciclos'].
?- trace(estado_final,+call), profundidad_con_ciclos(S).
 T Call: (9) estado_final(0)
 T Call: (10) estado_final(1)
 T Call: (11) estado_final(2)
...
```

  - **Por búsqueda en anchura**

```
?- ['paseo','b-anchura'].
?- trace(estado_final,+call), anchura(S).
 T Call: (10) estado_final(0)
 T Call: (11) estado_final(1)
 T Call: (12) estado_final(-1)
 T Call: (13) estado_final(2)
 T Call: (14) estado_final(-2)
 T Call: (15) estado_final(3)
 T Call: (16) estado_final(-3)
S = [0, -1, -2, -3]
Yes
```

Búsqueda en anchura

- **Procedimiento de búsqueda en achura**
  - **Un *nodo* es una lista de estados $[E_n, \dots, E_1]$ de forma que E_1 es el estado inicial y E_{i+1} es un sucesor de E_i.**
  - ***Abiertos* es la lista de nodos pendientes de analizar.**
  - `anchura(?S)` **se verifica si** `S` **es una solución del problema mediante búsqueda en anchura.**

```
anchura(S) :-
   estado_inicial(E),
   anchura([[E]],S).

anchura([[E|C]|_],S) :-
   estado_final(E),
   reverse([E|C],S).
anchura([N|R],S) :-
   expande([N|R],Sucesores),
   append(R,Sucesores,NAbiertos),
   anchura(NAbiertos,S).
```

Búsqueda en anchura

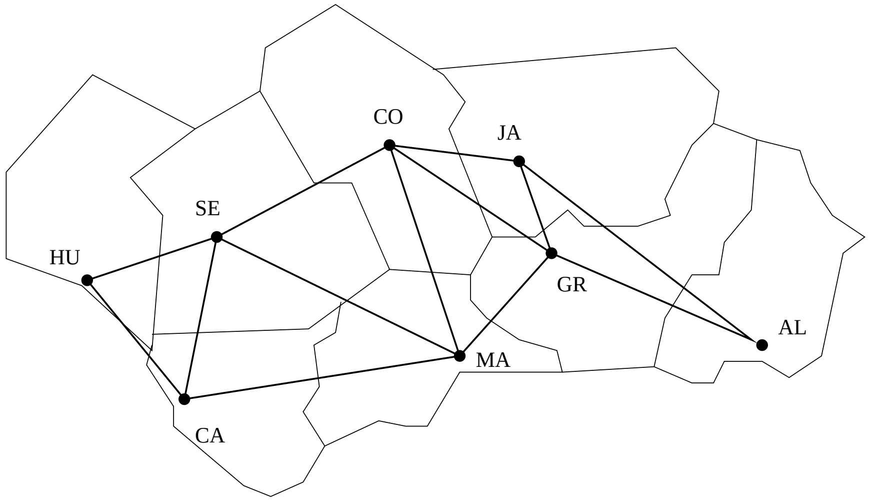
- `expande(+Abiertos,?Sucesores)` se verifica si `Sucesores` es la lista de los sucesores del primer elemento de `Abiertos` que no pertenecen ni al camino que lleva a dicho elemento ni a `Abiertos`. Por ejemplo,

```
? [jarras].
?- expande([[0-0]], L1).
L1 = [[4-0, 0-0], [0-3, 0-0]]
?- expande([[4-0, 0-0], [0-3, 0-0]], L2).
L2 = [[4-3, 4-0, 0-0], [1-3, 4-0, 0-0]]
```

```
expande([[E|C]|R],Sucesores) :-
   findall([E1,E|C],
           (sucesor(E,E1),
            not(memberchk(E1,C)),
            not(memberchk([E1|_],[[E|C]|R]))),
           Sucesores).
```

Búsqueda óptima: Problema del viaje

- **Enunciado:**
  - **Nos encontramos en una capital andaluza (p.e. Sevilla) y deseamos ir a otra capital andaluza (p.e. Almería). Los autobuses sólo van de cada capital a sus vecinas.**



- **Solución:**

```
?- ['viaje', 'b-profundidad-con-ciclos', 'b-anchura'].
?- profundidad_con_ciclos(S).
S = [sevilla, córdoba, jaén, granada, almería]
?- anchura(S).
S = [sevilla, córdoba, granada, almería]
```

Búsqueda óptima: Problema del viaje

- **Representación (`viaje.pl`)**
  - `estado_inicial(?E)` **se verifica si** `E` **es el estado inicial.**

    ```
    estado_inicial(sevilla).
    ```

  - `estado_final(?E)` **se verifica si** `E` **es un estado final.**

    ```
    estado_final(almería).
    ```

  - `sucesor(+E1,?E2)` **se verifica si** `E2` **es un sucesor del estado** `E1`**.**

    ```
    sucesor(E1,E2) :-
       ( conectado(E1,E2) ; conectado(E2,E1) ).
    ```

  - `conectado(?E1,?E2)` **se verifica si** `E1` **y** `E2` **están conectados.**

    ```
    conectado(huelva,sevilla).        conectado(huelva,cádiz).
    conectado(sevilla,córdoba).       conectado(sevilla,málaga).
    conectado(sevilla,cádiz).         conectado(córdoba,jaén).
    conectado(córdoba,granada).       conectado(córdoba,málaga).
    conectado(cádiz,málaga).          conectado(málaga,granada).
    conectado(jaén,granada).          conectado(granada,almería).
    ```

Búsqueda óptima: Problema del viaje

- `coste(+E1,+E2,?C)` **se verifica si** `C` **es la distancia entre** `E1` **y** `E2`**.**

```
coste(E1,E2,C) :-
   ( distancia(E1,E2,C) ; distancia(E2,E1,C) ).
```

- `distancia(+E1,+E2,?D)` **se verifica si** `D` **es la distancia de** `E1` **a** `E2`**.**

```
distancia(huelva,sevilla,94).
distancia(huelva,cádiz,219).
distancia(sevilla,córdoba,138).
distancia(sevilla,málaga,218).
distancia(sevilla,cádiz,125).
distancia(córdoba,jaén,104).
distancia(córdoba,granada,166).
distancia(córdoba,málaga,187).
distancia(cádiz,málaga,265).
distancia(málaga,granada,129).
distancia(jaén,granada,99).
distancia(granada,almería,166).
```

Búsqueda óptima

- **Búsqueda óptima (`b-optima-1.pl`)**

  - `óptima_1(?S)` **se verifica si** `S` **es una solución óptima del problema; es decir,** `S` **es una solución del problema y no hay otra solución de menor coste.**

```
óptima_1(S) :-
   profundidad_con_ciclos(S),
   coste_camino(S,C),
   not((profundidad_con_ciclos(S1),
        coste_camino(S1,C1),
        C1 < C)).
```

  - `coste_camino(+L,?C)` **se verifica si** `C` **es el coste del camino** `L`**.**

```
coste_camino([E2,E1],C) :-
   coste(E2,E1,C).
coste_camino([E2,E1|R],C) :-
   coste(E2,E1,C1),
   coste_camino([E1|R],C2),
   C is C1+C2.
```

Búsqueda óptima (II)

- **2^o procedimiento de búsqueda óptima (`b-optima-2.pl`)**
  - **Un *nodo* es un término de la forma $C - [E_n, \ldots, E_1]$ tal que E_1 es el estado inicial y E_{i+1} es un sucesor de E_i y C es el coste de dicho camino.**
  - `óptima(?S)` **se verifica si** `S` **es una solución del problema mediante búsqueda óptima; es decir,** `S` **es la solución obtenida por búsqueda óptima a partir de** `[0-[E]]`**, donde** `E` **el estado inicial.**

```
óptima(S) :-
   estado_inicial(E),
   óptima([0-[E]],S).

óptima([_C-[E|R]|_RA],S) :-
   estado_final(E),
   reverse([E|R],S).
óptima([N|RAbiertos],S) :-
   expande(N,Sucesores),
   append(RAbiertos,Sucesores,Abiertos1),
   sort(Abiertos1,Abiertos2),
   óptima(Abiertos2,S).
```

Búsqueda óptima (II)

- `expande(+N,?Sucesores)` se verifica si `Sucesores` es la lista de sucesores del nodo `N` (es decir, si `N=C-[E|R]`, entonces `Sucesores` son los nodos de la forma `C1-[E1,E|R]` donde `E1` es un sucesor de `E` que no pertenece a `[E|R]` y `C1` es la suma de `C` y el coste de `E` a `E1`).

```
expande(C-[E|R],Sucesores) :-
   findall(C1-[E1,E|R],
           (sucesor(E,E1),
            not(member(E1,[E|R])),
            coste(E,E1,C2),
            C1 is C+C2),
           Sucesores).
```

- **Comparaciones**

```
?- time(óptima_1(S)).
% 1,409 inferences in 0.00 seconds (Infinite Lips)
S = [sevilla, córdoba, granada, almería]

?-  time(óptima(S)).
% 907 inferences in 0.00 seconds (Infinite Lips)
S = [sevilla, córdoba, granada, almería]
```

Bibliografía

- Bratko, I. *Prolog Programming for Artificial Intelligence (2nd ed.)* (Addison–Wesley, 1990)
  - Cap. 11 "Basic problem–solving strategies"
- Flach, P. *Simply Logical (Intelligent Reasoning by Example)* (John Wiley, 1994)
  - Cap. 5: "Seaching graphs"
- Poole, D.; Mackworth, A. y Goebel, R. *Computational Intelligence (A Logical Approach)* (Oxford University Press, 1998)
  - Cap. 4: "Searching"
- Shoham, Y. *Artificial Intelligence Techniques in Prolog* (Morgan Kaufmann, 1994)
  - Cap. 2 "Search"