Tema 3:
Resolución de problemas mediante búsqueda sin información

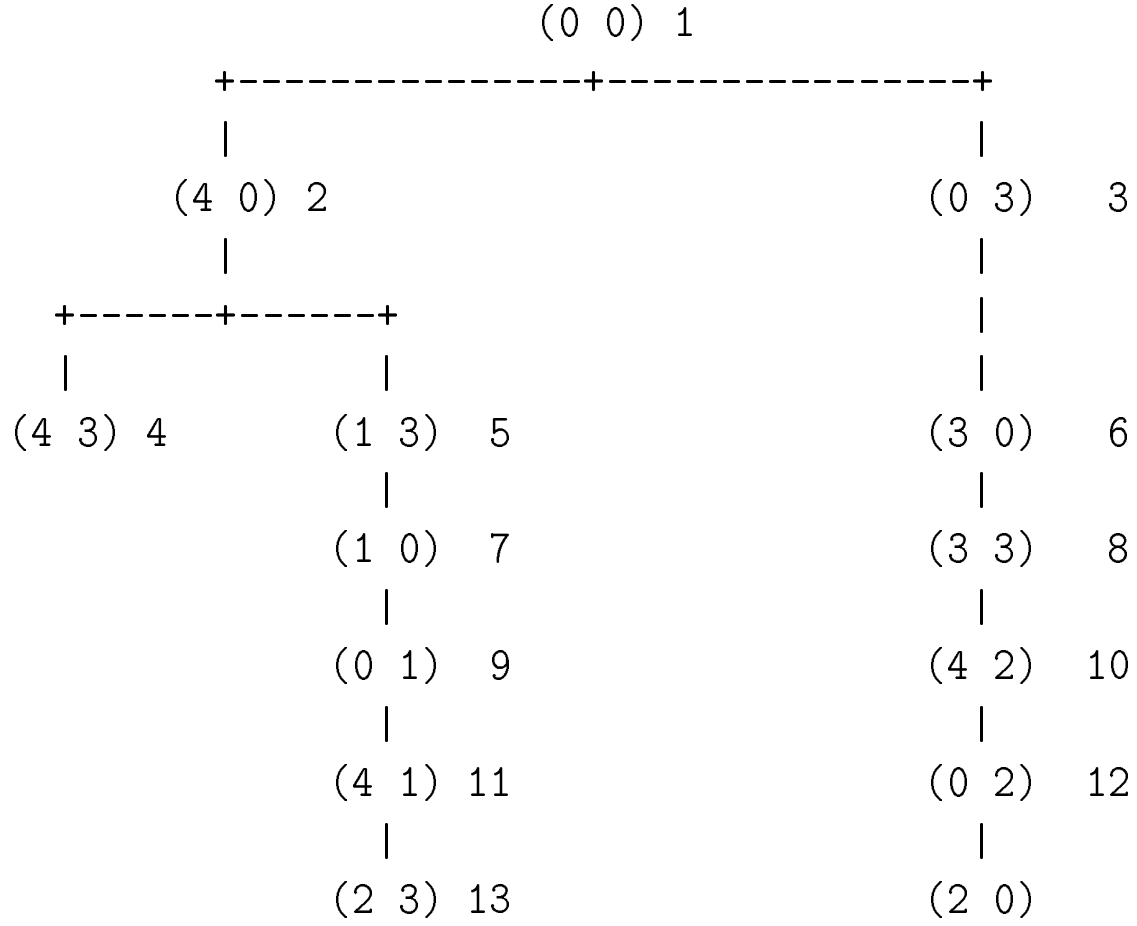
Problema de las jarras de agua

- 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 de estados: `(x y)`, `x` $\in \{0,1,2,3,4\}$, `y` $\in \{0,1,2,3\}$
- Número de estados $= 20$

Problema de las jarras de agua

- Estado inicial: `(0 0)`
- Estados finales: `(2 n)`
- Operadores:
  - Llenar la jarra de 3 litros con la bomba.
  - Llenar la jarra de 4 litros con la bomba.
  - Llenar la jarra de 3 litros con la jarra de 4 litros.
  - Llenar la jarra de 4 litros con la jarra de 3 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.
  - Vaciar la jarra de 3 litros en el suelo.
  - Vaciar la jarra de 4 litros en el suelo.

Búsqueda en anchura para el problema de las jarra



Búsqueda en anchura para el problema de las jarras

Nodo	Actual	Sucesores	Abiertos
1	(0 0)	((4 0) (0 3))	((4 0) (0 3))
2	(4 0)	((4 3) (1 3))	((0 3) (4 3) (1 3))
3	(0 3)	((3 0))	((4 3) (1 3) (3 0))
4	(4 3)	()	((1 3) (3 0))
5	(1 3)	((1 0))	((3 0) (1 0))
6	(3 0)	((3 3))	((1 0) (3 3))
7	(1 0)	((0 1))	((3 3) (0 1))
8	(3 3)	((4 2))	((0 1) (4 2))
9	(0 1)	((4 1))	((4 2) (4 1))
10	(4 2)	((0 2))	((4 1) (0 2))
11	(4 1)	((2 3))	((0 2) (2 3))
12	(0 2)	((2 0))	((2 3) (2 0))
13	(2 3)		

Implementación de nodos

```
(defstruct (nodo (:constructor crea-nodo)
                 (:conc-name nil)
  estado
  camino)
```

Procedimiento de búsqueda en anchura

1. Crear las siguientes variables locales
   1.1. ABIERTOS (para almacenar los nodos generados aun no analizados) con valor la lista formado por el nodo inicial (es decir, el nodo cuyo estado es el estado inicial y cuyo camino es la lista vacia);
   1.2. CERRADOS (para almacenar los nodos analizados) con valor la lista vacia;
   1.3. ACTUAL (para almacenar el nodo actual) con valor la lista vacia.
   1.4. NUEVOS-SUCESORES (para almacenar la lista de los sucesores del nodo actual que aun no se han generado) con valor la lista vacia.

Procedimiento de búsqueda en anchura

```
2. Mientras que ABIERTOS no esta vacia,
   2.1 Hacer ACTUAL el primer nodo de ABIERTOS
   2.2 Hacer ABIERTOS el resto de ABIERTOS
   2.3 Poner el nodo ACTUAL en CERRADOS.
   2.4 Si el nodo ACTUAL es un final,
       2.4.1 devolver el nodo ACTUAL y terminar.
       2.4.2 en caso contrario, hacer
             2.4.2.1 NUEVOS-SUCESORES la lista de sucesores del nodo
                     ACTUAL que no estan en ABIERTOS ni en CERRADOS y
             2.4.2.2 ABIERTOS la lista obtenida anadiendo los
                     NUEVOS-SUCESORES al final de ABIERTOS.
3. Si ABIERTOS esta vacia, devolver NIL.
```

Lisp

```
* (SYMBOL-VALUE SIMBOLO)
      (setf dos 3)                  =>  3
      (symbol-value 'dos)           =>  3
      (setf (symbol-value 'dos) 4)  =>  4
      dos                           =>  4

* ((LAMBDA (VAR-1...VAR-N) E-1 ... E-M) VAL-1 ... VAL-N)
      ((lambda (m n) (+ m n)) 2 3)  =>  5
```

Lisp

```
* (SYMBOL-FUNCTION SIMBOLO)
      > (setf (symbol-function ´cuadrado)
              ´(lambda (x) (* x x)))
      (LAMBDA (X) (* X X)
      > (cuadrado 3)
      9
      > (symbol-function ´cuadrado)
      (LAMBDA (X) (* X X)
      > (defun cubo (x) (* x x x))
      CUBO
      > (cubo 3)
      27
      > (symbol-function ´cubo)
      (LAMBDA-BLOCK CUBO (X) (* X X X))
```

Implementación de la búsqueda en anchura

- Funciones y variables dependientes del problema
  - `*ESTADO-INICIAL*`
  - `(ES-ESTADO-FINAL ESTADO)`
  - `*OPERADORES*`
  - `(<OPERADOR> ESTADO)`

Implementación de la búsqueda en anchura

```
(defun busqueda-en-anchura ()
  (let ((abiertos (list (crea-nodo :estado *estado-inicial*          ;1.1
                                   :camino nil)))
        (cerrados nil)                                               ;1.2
        (actual nil)                                                 ;1.3
        (nuevos-sucesores nil))                                      ;1.4
    (loop until (null abiertos) do                                   ;2
          (setf actual (first abiertos))                             ;2.1
          (setf abiertos (rest abiertos))                            ;2.2
          (setf cerrados (cons actual cerrados))                     ;2.3
          (cond ((es-estado-final (estado actual))                   ;2.4
                 (return actual))                                    ;2.4.1
                (t (setf nuevos-sucesores                            ;2.4.2.1
                         (nuevos-sucesores actual abiertos cerrados))
                   (setf abiertos                                    ;2.4.2.2
                         (append abiertos nuevos-sucesores)))))))
```

Implementación de la búsqueda en anchura

```
(defun nuevos-sucesores (nodo abiertos cerrados)
  (elimina-duplicados (sucesores nodo) abiertos cerrados))

(defun sucesores (nodo)
  (loop for operador in *operadores*
        when (nodo-sucesor nodo operador)
        collect (nodo-sucesor nodo operador)))

(defun nodo-sucesor (nodo operador)
  (let ((siguiente-estado (funcall (symbol-function operador)          ;1
                                   (estado nodo))))
    (when siguiente-estado                                             ;2
          (crea-nodo :estado siguiente-estado                          ;2.1
                     :camino (cons operador                            ;2.2
                                   (camino nodo))))))
```

Implementación de la búsqueda en anchura

```
(defun elimina-duplicados (nodos abiertos cerrados)
  (loop for nodo in nodos
        when (and (not (esta nodo abiertos))
                  (not (esta nodo cerrados)))
        collect nodo))

(defun esta (nodo lista-de-nodos)
  (let ((estado (estado nodo)))
    (loop for n in lista-de-nodos
          thereis (equalp estado (estado n)))))
```

Solución de las jarras por búsqueda en anchura

```
> (load "../tema-02/jarras-1.lsp")
T
> (load "anchura.lsp")
T
> (busqueda-en-anchura)
#S(NODO :ESTADO (2 3)
        :CAMINO (LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4
                 VACIAR-JARRA-4-EN-JARRA-3
                 VACIAR-JARRA-3
                 LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4))
```

Búsqueda en anchura para el problema de las jarras

```
> (trace es-estado-final)
(ES-ESTADO-FINAL)
> (busqueda-en-anchura)
1. Trace: (ES-ESTADO-FINAL '(0 0))
1. Trace: (ES-ESTADO-FINAL '(4 0))
1. Trace: (ES-ESTADO-FINAL '(0 3))
1. Trace: (ES-ESTADO-FINAL '(4 3))
1. Trace: (ES-ESTADO-FINAL '(1 3))
1. Trace: (ES-ESTADO-FINAL '(3 0))
1. Trace: (ES-ESTADO-FINAL '(1 0))
1. Trace: (ES-ESTADO-FINAL '(3 3))
1. Trace: (ES-ESTADO-FINAL '(0 1))
1. Trace: (ES-ESTADO-FINAL '(4 2))
1. Trace: (ES-ESTADO-FINAL '(4 1))
1. Trace: (ES-ESTADO-FINAL '(0 2))
1. Trace: (ES-ESTADO-FINAL '(2 3))
```

Soluciones de los problemas en anchura

```
> (load "anchura.lsp")
T
> (load "../tema-02/viaje.lsp")
T
> (time (busqueda-en-anchura))

Real time: 0.202387 sec.
Run time: 0.2 sec.
Space: 4700 Bytes
#S(NODO :ESTADO ALMERIA
        :CAMINO (IR-A-ALMERIA
                 IR-A-GRANADA
                 IR-A-CORDOBA))
```

Soluciones de los problemas en anchura

```
> (load "../tema-02/granjero-1.lsp")
T
> (time (busqueda-en-anchura))
Real time: 0.214693 sec.
Run time: 0.21 sec.
Space: 5180 Bytes
#S(NODO :ESTADO (D D D D)
        :CAMINO (PASAN-GRANJERO-Y-CABRA
                 PASA-GRANJERO-SOLO
                 PASAN-GRANJERO-Y-COL
                 PASAN-GRANJERO-Y-CABRA
                 PASAN-GRANJERO-Y-LOBO
                 PASA-GRANJERO-SOLO
                 PASAN-GRANJERO-Y-CABRA))
```

Soluciones de los problemas en anchura

```
> (load "../tema-02/jarras-1.lsp")
T
> (time (busqueda-en-anchura))
Real time: 0.455523 sec.
Run time: 0.46 sec.
Space: 10428 Bytes
#S(NODO :ESTADO (2 3)
        :CAMINO (LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4
                 VACIAR-JARRA-4-EN-JARRA-3
                 VACIAR-JARRA-3
                 LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4))
```

Soluciones de los problemas en anchura

```
> (load "../tema-02/8-puzzle.lsp")
T
> (time (busqueda-en-anchura))
Real time: 5.640544 sec.
Run time: 5.64 sec.
Space: 86348 Bytes
#S(NODO :ESTADO #2A((1 2 3) (8 H 4) (7 6 5))
        :CAMINO (MOVER-DERECHA
                 MOVER-ABAJO
                 MOVER-IZQUIERDA
                 MOVER-ARRIBA
                 MOVER-ARRIBA))
```

Soluciones de los problemas en anchura

	Tiempo (seg.)	Espacio (bytes)	Nodos analizados	Maximo en abiertos	Profundidad maxima
Viaje	0.2	4.700	8	4	3
Granjero	0.21	5.180	10	2	7
Jarras	0.46	10.428	13	3	6
8-puzzle	5.64	86.348	46	33	5

Limitaciones de la búsqueda en anchura

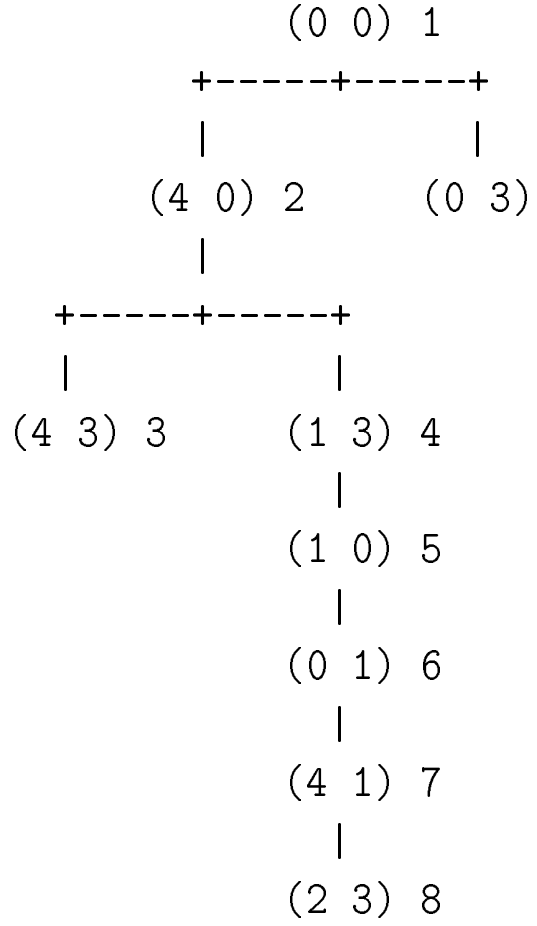
```
> (load "../tema-02/8-puzzle.lsp")
T
> (load "anchura.lsp")
T
> (setf *estado-inicial*
        (make-array '(3 3)
                    :initial-contents '((4 8 1) (3 H 2) (7 6 5))))
#2A((4 8 1) (3 H 2) (7 6 5))
> (time (busqueda-en-anchura))

** - EVAL: User break
Real time: 100.43055 sec.
Run time: 96.08 sec.
Space: 1457680 Bytes
GC: 3, GC time: 0.47 sec.
```

Propiedades de la búsqueda en anchura

- Complejidad:
  - r = **factor de ramificación**
  - p = **profundidad de la solución**
  - **Complejidad en tiempo:** $O(r^p)$
  - **Complejidad en espacio:** $O(r^p)$
- Completa
- Optimal

Búsqueda en profundidad para el problema de las jarras



Búsqueda en profundidad para el problema de las jarras

Nodo	Actual	Sucesores	Abiertos
1	(0 0)	((4 0) (0 3))	((4 0) (0 3))
2	(4 0)	((4 3) (1 3))	((4 3) (1 3) (0 3))
3	(4 3)	()	((1 3) (0 3))
4	(1 3)	((1 0))	((1 0) (0 3))
5	(1 0)	((0 1))	((0 1) (0 3))
6	(0 1)	((4 1))	((4 1) (0 3))
7	(4 1)	((2 3))	((2 3) (0 3))
8	(2 3)		

- Estados de la solución: ((2 3) (4 1) (0 1) (1 0) (1 3) (4 0) (0 0))

Procedimiento de búsqueda en profundidad

1. Crear las siguientes variables locales
   1.1. ABIERTOS (para almacenar los nodos generados aun no analizados) con valor la lista formado por el nodo inicial (es decir, el nodo cuyo estado es el estado inicial y cuyo camino es la lista vacia);
   1.2. CERRADOS (para almacenar los nodos analizados) con valor la lista vacia;
   1.3. ACTUAL (para almacenar el nodo actual) con valor la lista vacia.
   1.4. NUEVOS-SUCESORES (para almacenar la lista de los sucesores del nodo actual que aun no se han generado) con valor la lista vacia.

Procedimiento de búsqueda en profundidad

```
2. Mientras que ABIERTOS no esta vacia,
   2.1 Hacer ACTUAL el primer nodo de ABIERTOS
   2.2 Hacer ABIERTOS el resto de ABIERTOS
   2.3 Poner el nodo ACTUAL en CERRADOS.
   2.4 Si el nodo ACTUAL es un objetivo,
       2.4.1 devolver el nodo ACTUAL y terminar.
       2.4.2 en caso contrario, hacer
             2.4.2.1 NUEVOS-SUCESORES la lista de sucesores del nodo
                     ACTUAL que no estan en ABIERTOS ni en CERRADOS y
             2.4.2.2 ABIERTOS la lista obtenida anadiendo los
                     NUEVOS-SUCESORES al principio de ABIERTOS.
3. Si ABIERTOS esta vacia, devolver NIL.
```

Implementación de la búsqueda en profundidad

```
(defun busqueda-en-profundidad ()
  (let ((abiertos (list (crea-nodo :estado *estado-inicial*          ;1.1
                                   :camino nil)))
        (cerrados nil)                                               ;1.2
        (actual nil)                                                 ;1.3
        (nuevos-sucesores nil))                                      ;1.4
    (loop until (null abiertos) do                                   ;2
          (setf actual (first abiertos))                             ;2.1
          (setf abiertos (rest abiertos))                            ;2.2
          (setf cerrados (cons actual cerrados))                     ;2.3
          (cond ((es-estado-final (estado actual))                   ;2.4
                 (return actual))                                    ;2.4.1
                (t (setf nuevos-sucesores                            ;2.4.2.1
                         (nuevos-sucesores actual abiertos cerrados))
                   (setf abiertos                                    ;2.4.2.2
                         (APPEND NUEVOS-SUCESORES ABIERTOS)))))))
```

Solución de las jarras por búsqueda en profundidad

```
> (load "../tema-02/jarras-1.lsp")
T
> (load "profundidad")
T
> (busqueda-en-profundidad)
#S(NODO :ESTADO (2 3)
        :CAMINO (LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4
                 VACIAR-JARRA-4-EN-JARRA-3
                 VACIAR-JARRA-3
                 LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4))
```

Búsqueda en profundidad para el problema de las jarras

```
> (trace es-estado-final)
(ES-ESTADO-FINAL)
> (busqueda-en-profundidad)

1. Trace: (ES-ESTADO-FINAL '(0 0))
1. Trace: (ES-ESTADO-FINAL '(4 0))
1. Trace: (ES-ESTADO-FINAL '(4 3))
1. Trace: (ES-ESTADO-FINAL '(1 3))
1. Trace: (ES-ESTADO-FINAL '(1 0))
1. Trace: (ES-ESTADO-FINAL '(0 1))
1. Trace: (ES-ESTADO-FINAL '(4 1))
1. Trace: (ES-ESTADO-FINAL '(2 3))
```

Búsqueda en profundidad para el problema de las jarras

```
#S(NODO :ESTADO (2 3)
        :CAMINO (LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4
                 VACIAR-JARRA-4-EN-JARRA-3
                 VACIAR-JARRA-3
                 LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4))
```

Soluciones de los problemas en profundidad

```
> (load "profundidad.lsp")
T
> (load "../tema-02/viaje.lsp")
T
> (time (busqueda-en-profundidad))

Real time: 0.118671 sec.
Run time: 0.12 sec.
Space: 2868 Bytes
#S(NODO :ESTADO ALMERIA
        :CAMINO (IR-A-ALMERIA
                 IR-A-GRANADA
                 IR-A-CORDOBA))
```

Soluciones de los problemas en profundidad

```
> (load "../tema-02/granjero-1.lsp")
T
> (time (busqueda-en-profundidad))

Real time: 0.200578 sec.
Run time: 0.18 sec.
Space: 4180 Bytes
#S(NODO :ESTADO (D D D D)
        :CAMINO (PASAN-GRANJERO-Y-CABRA
                 PASA-GRANJERO-SOLO
                 PASAN-GRANJERO-Y-COL
                 PASAN-GRANJERO-Y-CABRA
                 PASAN-GRANJERO-Y-LOBO
                 PASA-GRANJERO-SOLO
                 PASAN-GRANJERO-Y-CABRA))
```

Soluciones de los problemas en profundidad

```
> (load "../tema-02/jarras-1.lsp")
T
> (time (busqueda-en-profundidad))

Real time: 0.24973 sec.
Run time: 0.23 sec.
Space: 5324 Bytes
#S(NODO :ESTADO (2 3)
        :CAMINO (LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4
                 VACIAR-JARRA-4-EN-JARRA-3
                 VACIAR-JARRA-3
                 LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4))
```

Soluciones de los problemas en profundidad

	Tiempo (seg.)	Espacio (bytes)	Nodos analizados	Maximo en abiertos	Profundidad maxima
Viaje	0.11	2.868	5	4	3
Granjero	0.18	4.180	8	3	7
Jarras	0.23	5.324	8	3	6
8-puzzle	>>	>>	>>	>>	>>

Solución del 8-puzzle (II) en profundidad

```
> (load "../tema-02/8-puzzle.lsp")
T
> (load "profundidad.lsp")
T
> (setf *estado-inicial*
        (make-array '(3 3)
                    :initial-contents '((4 8 1) (3 H 2) (7 6 5))))
#2A((4 8 1) (3 H 2) (7 6 5))
> (time (busqueda-en-profundidad))
Real time: 1.004533 sec.
Run time: 1.01 sec.
Space: 15692 Bytes
#S(NODO :ESTADO #2A((1 2 3) (8 H 4) (7 6 5))
        :CAMINO MOVER-IZQUIERDA MOVER-IZQUIERDA MOVER-ABAJO MOVER-DERECHA
                MOVER-DERECHA MOVER-ARRIBA MOVER-IZQUIERDA))
```

Propiedades de la búsqueda en profundidad

- Complejidad:
  - r = **factor de ramificación**
  - m = **máxima profundidad de la búsqueda**
  - **Complejidad en tiempo:** $O(r^m)$
  - **Complejidad en espacio:** $O(rm)$
- No es completa.
- No es optimal

Lisp: Argumentos claves

```
(defun f (&key (x 1) (y 2)) (list x y))  =>  F
(f :x 5 :y 3)                            =>  (5 3)
(f :y 3 :x 5)                            =>  (5 3)
(f :y 3)                                 =>  (1 3)
(f)                                      =>  (1 2)
```

Búsqueda en profundidad acotada

```
(defun busqueda-en-profundidad-acotada (&key (COTA 5))
  (let ((abiertos (list (crea-nodo :estado *estado-inicial* :camino nil))) ;1.1
        (cerrados nil)                                                     ;1.2
        (actual nil)                                                       ;1.3
        (nuevos-sucesores nil))                                            ;1.4
    (loop until (null abiertos) do                                         ;2
          (setf actual (first abiertos))                                   ;2.1
          (setf abiertos (rest abiertos))                                  ;2.2
          (setf cerrados (cons actual cerrados))                           ;2.3
          (cond ((es-estado-final (estado actual))                         ;2.4
                 (return actual))                                          ;2.4.1
                ((< (LENGTH (CAMINO ACTUAL)) COTA)
                 (setf nuevos-sucesores                                    ;2.4.2.1
                       (nuevos-sucesores actual abiertos cerrados))
                 (setf abiertos                                            ;2.4.2.2
                       (append nuevos-sucesores abiertos)))))))
```

Solución del 8-puzzle por profundidad acotada

```
> (load "../tema-02/8-puzzle.lsp")
T
> (load "profundidad-acotada.lsp")
T
> (time (busqueda-en-profundidad-acotada))

Real time: 1.704729 sec.
Run time: 1.68 sec.
Space: 25252 Bytes
#S(NODO :ESTADO #2A((1 2 3) (8 H 4) (7 6 5))
        :CAMINO (MOVER-DERECHA
                 MOVER-ABAJO
                 MOVER-IZQUIERDA
                 MOVER-ARRIBA
                 MOVER-ARRIBA))
```

Solución del 8-puzzle por profundidad acotada

```
> (setf *estado-inicial*
        (make-array '(3 3)
                    :initial-contents '((4 8 1) (3 H 2) (7 6 5))))
#2A((4 8 1) (3 H 2) (7 6 5))
> (time (busqueda-en-profundidad-acotada))

Real time: 4.24191 sec.
Run time: 4.22 sec.
Space: 61188 Bytes
NIL
```

Solución del 8-puzzle por profundidad acotada

```
> (time (busqueda-en-profundidad-acotada :cota 12))

Real time: 1.042679 sec.
Run time: 1.01 sec.
Space: 15788 Bytes
#S(NODO :ESTADO #2A((1 2 3) (8 H 4) (7 6 5))
        :CAMINO (MOVER-IZQUIERDA   MOVER-ABAJO
                 MOVER-DERECHA     MOVER-DERECHA
                 MOVER-ARRIBA      MOVER-IZQUIERDA
                 MOVER-IZQUIERDA   MOVER-ABAJO
                 MOVER-DERECHA     MOVER-DERECHA
                 MOVER-ARRIBA      MOVER-IZQUIERDA))
```

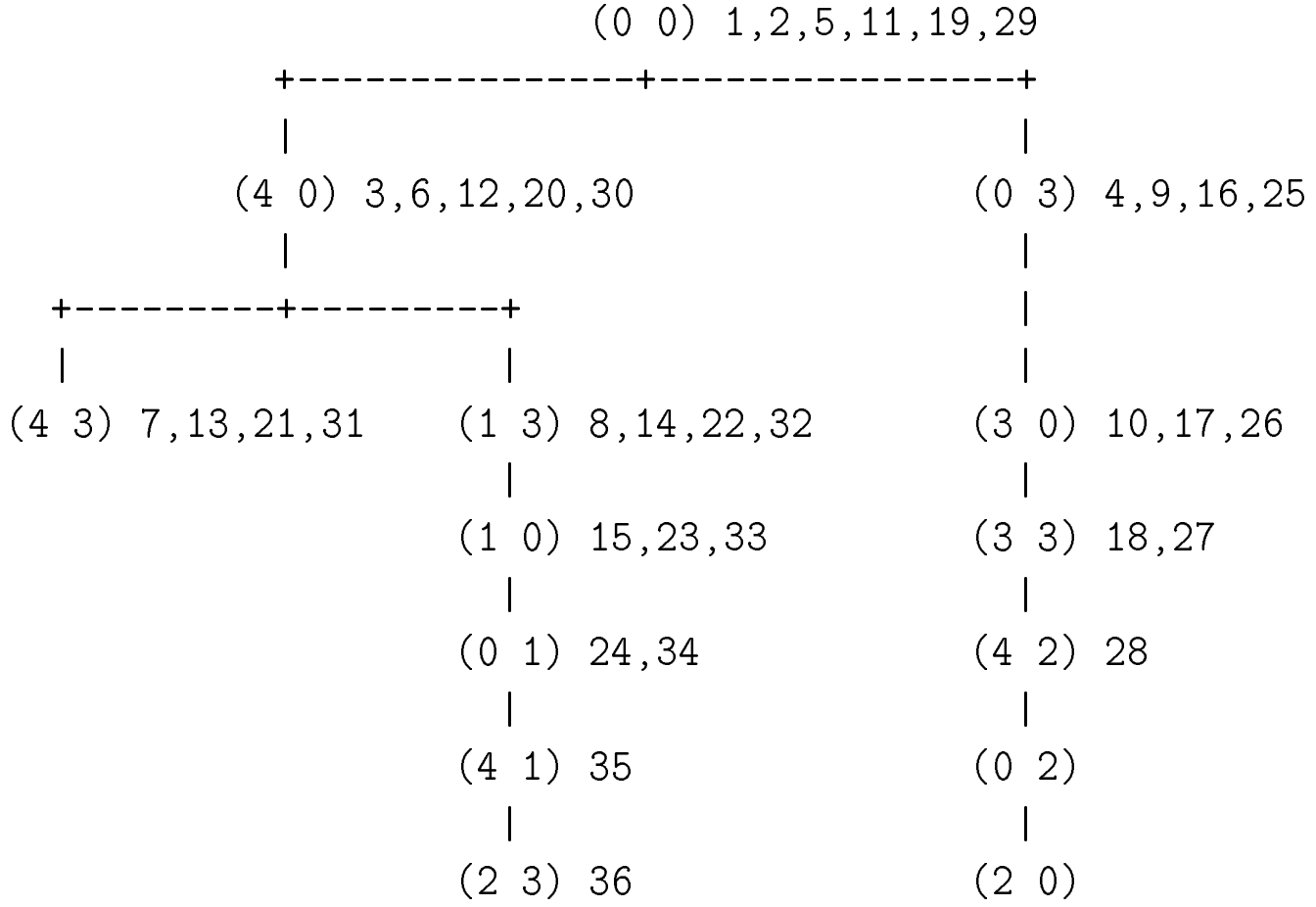
Comparación de soluciones del 8-puzzle

	Anchura		Profundidad		Profundidad cota = 5		Profundidad cota = 12	
Estado inicial	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)
283 164 7 5	5.64	86.348	>>	>>	1.68	25.252	76.17	1.044.436
481 3 2 765	>>	>>	1.01	15.692	>>	>>	1.01	15.788

Propiedades de la búsqueda en profundidad acotada

- Complejidad:
  - r = factor de ramificación
  - c = cota de la profundidad
  - Complejidad en tiempo: $O(r^c)$
  - Complejidad en espacio: $O(rc)$
- Es completa cuando $c \geq p$ (p = profundidad de solución)
- No es optimal

Profundidad iterativa para el problema de las jarras



Procedimiento de búsqueda en profundidad iterativa

```
(defun busqueda-en-profundidad-iterativa (&key (cota-inicial 5))
  (loop for n from cota-inicial
        thereis (busqueda-en-profundidad-acotada :cota n)))
```

Jarras por búsqueda en profundidad iterativa

```
> (load "../tema-02/jarras-1.lsp")
T
> (load "profundidad-iterativa.lsp")
T
> (time (busqueda-en-profundidad-iterativa :cota-inicial 0))
Real time: 1.112849 sec.
Run time: 1.11 sec.
Space: 26656 Bytes
#S(NODO :ESTADO (2 3)
        :CAMINO (LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4
                 VACIAR-JARRA-4-EN-JARRA-3
                 VACIAR-JARRA-3
                 LLENAR-JARRA-3-CON-JARRA-4
                 LLENAR-JARRA-4))
```

Comparación de soluciones del 8-puzzle

Estado inicial	Anchura		Profundidad		Profundidad iterativa	
	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)
283 164 7 5	5.64	86.348	>>	>>	1.68	25.252
481 3 2 765	>>	>>	1.01	15.692	424.81	5.802.744

Lisp: Compilación

```
> (compile-file "../tema-02/8-puzzle.lsp")

Compiling file /home/jalonso/hse-96/tema-02/8-puzzle.lsp ...

Compilation of file /home/jalonso/hse-96/tema-02/8-puzzle.lsp is finished.
0 errors, 0 warnings
T
> (load "../tema-02/8-puzzle")
T
> (compile-file "anchura.lsp")

Compiling file /home/jalonso/hse-96/tema-03/anchura.lsp ...

Compilation of file /home/jalonso/hse-96/tema-03/anchura.lsp is finished.
0 errors, 0 warnings
T
```

Lisp: Compilación

```
> (load "anchura")
T
> (time (busqueda-en-anchura))

Real time: 0.329973 sec.
Run time: 0.31 sec.
Space: 41540 Bytes
#S(NODO :ESTADO #2A((1 2 3) (8 H 4) (7 6 5))
        :CAMINO (MOVER-DERECHA
                 MOVER-ABAJO
                 MOVER-IZQUIERDA
                 MOVER-ARRIBA
                 MOVER-ARRIBA))
```

Comparación del 8-puzzle (con compilaciones)

Estado inicial	Anchura		Profundidad		Profundidad iterativa	
	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)
283 164 7 5	0.31	41.540	>>	>>	0.1	14.908
481 3 2 765	>>	>>	0.06	9.892	19.3	1.079.384

Propiedades de la búsqueda en profundidad iterativa

- Complejidad:
  - r = **factor de ramificación**
  - p = **profundidad de solución**
  - **Complejidad en tiempo:** $O(r^p)$
  - **Complejidad en espacio:** $O(rp)$
- Es completa
- Es optimal

Comparación de procedimientos

	Anchura	Profundidad	Profundidad acotada	Profundidad iterativa
Tiempo	$O(r^p)$	$O(r^m)$	$O(r^c)$	$O(r^p)$
Espacio	$O(r^p)$	$O(rm)$	$O(rc)$	$O(rp)$
Completa	**Sí**	**No**	**Sí, si** $c \geq p$	**Sí**
Optimal	**Sí**	**No**	**No**	**Sí**

- r = **factor de ramificación**
- p = **profundidad de la solución**
- m = **máxima profundidad de la búsqueda**
- c = **cota de la profundidad**

Bibliografía

- Borrajo–93 Cap. 4 "Búsqueda".
- Haton–91 Cap. 2 "Resolución de problemas".
- Rich–94 Cap. 2 "Problemas, espacios problema y búsqueda".
- Shirai–87 Cap. 3 "Técnicas de búsqueda".
- Winston–91 Cap. 19 "Ejemplos que involucran búsquedas".
- Winston–94 Cap. 4 "Redes y búsqueda básica".