Tema 3: Técnicas básicas de búsqueda

José A. Alonso Jiménez
Francisco J. Martín Mateos

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Problema de las jarras de agua

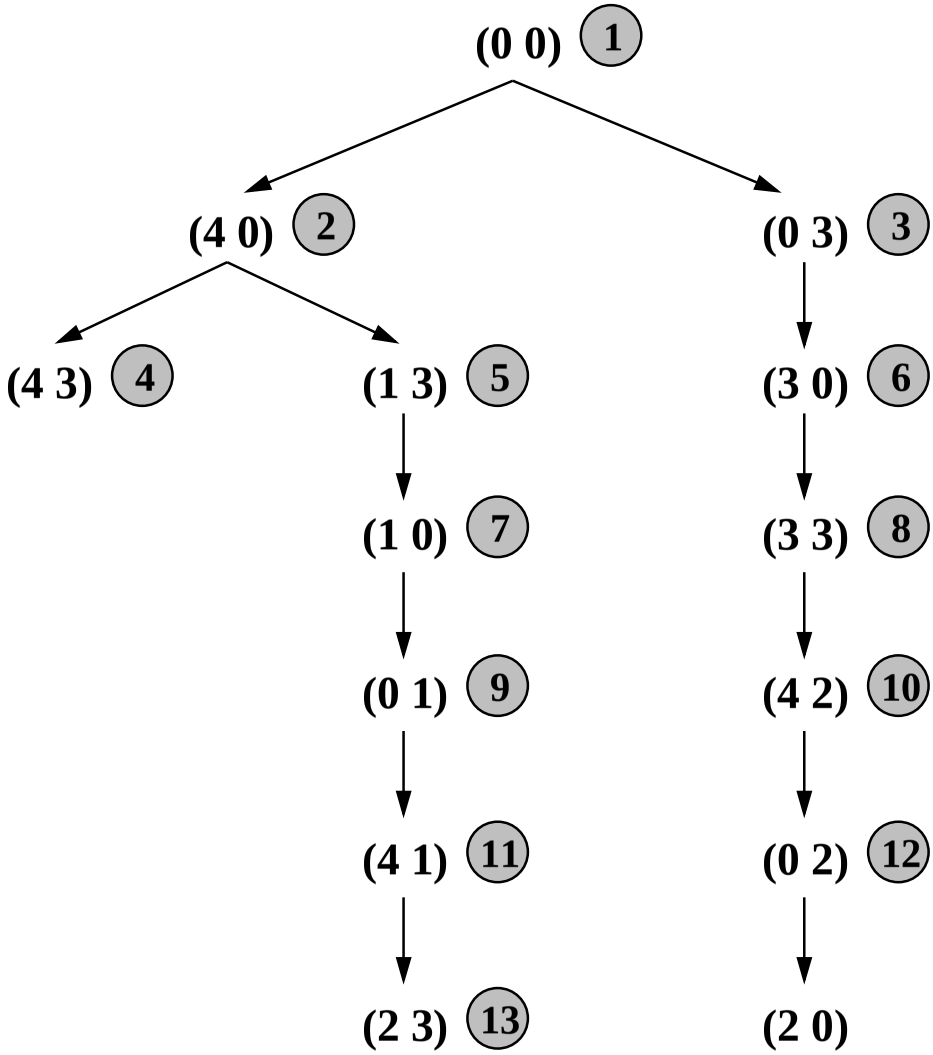
- **Enunciado:**
  - **Se tienen dos jarras, una de 4 litros de capacidad y otra de 3.**
  - **Ninguna de ellas tiene marcas de medición.**
  - **Se tiene una bomba que permite llenar las jarras de agua.**
  - **Averiguar cómo se puede lograr tener exactamente 2 litros de agua en la jarra de 4 litros de capacidad.**
- **Representación de estados:** $(\mathrm{x\ y}) \in \{0,1,2,3,4\} \times \{0,1,2,3\}$.
- **Número de estados: 20.**

Problema de las jarras de agua

- **Estado inicial: (0 0).**
- **Estados finales: (2 y).**
- **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.**

Problema de las jarras de agua

- **Grafo de búsqueda en anchura:**



Problema de las jarras de agua

- **Tabla de búsqueda en anchura:**

Nodo	Actual	Sucesores	Abiertos
			((0 0))
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)		

Definición de nodo

- **Nodo = Estado + Camino**
- **Representación de nodos en Lisp**

```
(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 aún no analizados) con valor la lista formada por el nodo inicial (es decir, el nodo cuyo estado es el estado inicial y cuyo camino es la lista vacía);
   1.2. CERRADOS (para almacenar los nodos analizados) con valor la lista vacía;
   1.3. ACTUAL (para almacenar el nodo actual) con valor la lista vacía.
   1.4. NUEVOS-SUCESORES (para almacenar la lista de los sucesores del nodo actual) con valor la lista vacía.

Procedimiento de búsqueda en anchura

```
2. Mientras que ABIERTOS no esté vacía,
   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 están en ABIERTOS ni en CERRADOS y
             2.4.2.2 ABIERTOS la lista obtenida añadiendo los
                     NUEVOS-SUCESORES al final de ABIERTOS.
3. Si ABIERTOS está vacía, devolver NIL.
```

Implementación de la búsqueda en anchura

- **Funciones y variables dependientes del problema:**
  - `*estado-inicial*`
  - `(es-estado-final estado)`
  - `*operadores*`
  - `(<operador> estado)`
  - `(aplica 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)
  (let ((resultado ()))
    (loop for operador in *operadores* do
          (let ((siguiente (sucesor nodo operador)))
               (when siguiente (push siguiente resultado))))
    (nreverse resultado)))

(defun sucesor (nodo operador)
  (let ((siguiente-estado (aplica operador (estado nodo))))
    (when siguiente-estado
          (crea-nodo :estado siguiente-estado
                     :camino (cons operador
                                   (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)))))
```

Soluciones de los problemas en anchura

- **Problema de las jarras:**

```
> (load "p-jarras-1.lsp")
T
> (load "b-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))
```

Soluciones de los problemas en anchura

```
> (trace 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))
> (untrace)
```

Soluciones de los problemas en anchura

```
> (time (busqueda-en-anchura))
Real time: 0.416386 sec.
Run time: 0.41 sec.
Space: 7236 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

- **Estadística de búsqueda en anchura:**

	Tiempo (seg.)	Espacio (bytes)	Nodos analizados	Máximo en abiertos	Profundidad máxima
Viaje	**0.18**	**3.260**	**8**	**4**	**3**
Granjero	**0.18**	**3.432**	**10**	**2**	**7**
Jarras	**0.41**	**7.236**	**13**	**3**	**6**
8-puzzle	**4.51**	**68.292**	**46**	**33**	**5**

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: $O(r^p)$.**
- **Es completa.**
- **Es minimal.**

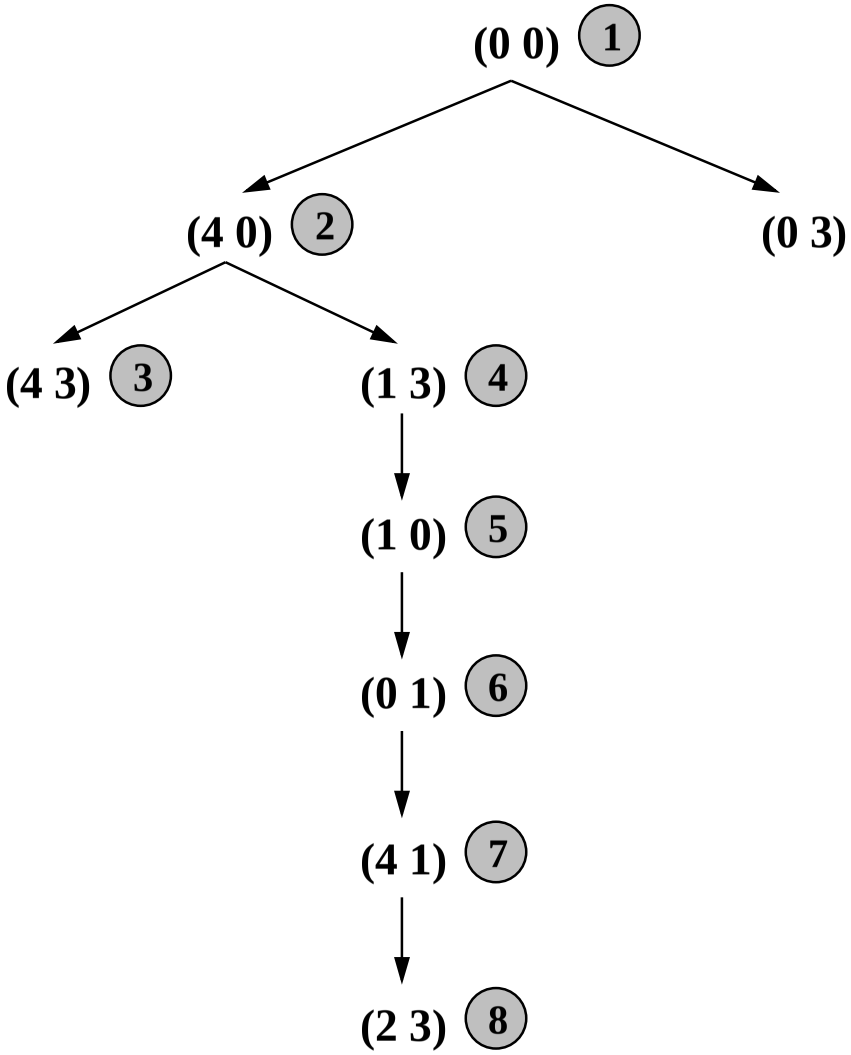
Limitaciones de la búsqueda en anchura

```
> (load "p-8-puzzle.lsp")
T
> (load "b-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.
```

Problema de las jarras de agua

- **Grafo de búsqueda en profundidad:**



Problema de las jarras de agua

- **Tabla de búsqueda en profundidad:**

Nodo	Actual	Sucesores	Abiertos
			((0 0))
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 aún no analizados) con valor la lista formada por el nodo inicial (es decir, el nodo cuyo estado es el estado inicial y cuyo camino es la lista vacía);
   1.2. CERRADOS (para almacenar los nodos analizados) con valor la lista vacía;
   1.3. ACTUAL (para almacenar el nodo actual) con valor la lista vacía.
   1.4. NUEVOS-SUCESORES (para almacenar la lista de los sucesores del nodo actual) con valor la lista vacía.

Procedimiento de búsqueda en profundidad

```
2. Mientras que ABIERTOS no esté vacía,
   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 están en ABIERTOS ni en CERRADOS y
             2.4.2.2 ABIERTOS la lista obtenida añadiendo los
                     NUEVOS-SUCESORES al principio de ABIERTOS.
3. Si ABIERTOS está vacía, 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)))))))
```

Soluciones de los problemas en profundidad

- **Problema de las jarras:**

```
> (load "p-jarras-1.lsp")
T
> (load "b-profundidad.lsp")
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))
```

Soluciones de los problemas en profundidad

```
> (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))
#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

- **Estadística de búsqueda en profundidad:**

	Tiempo (seg.)	Espacio (bytes)	Nodos analizados	Máximo en abiertos	Profundidad máxima
Viaje	**0.1**	**1.968**	**5**	**4**	**3**
Granjero	**0.14**	**2.800**	**8**	**3**	**7**
Jarras	**0.19**	**3.576**	**8**	**3**	**6**
8-puzzle	—	—	—	—	—

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 minimal.**

Problema sin solución en profundidad

- **Enunciado:**
  - **Una persona puede moverse en línea recta dando cada vez un paso hacia la derecha o hacia la izquierda.**
  - **Representamos su posición mediante un número entero.**
  - **La posición inicial es 0.**
  - **La posición aumenta en 1 por cada paso a la derecha.**
  - **La posición decrece en 1 por cada paso a la izquierda.**
  - **El problema consiste en llegar a la posición -3.**

Problema sin solución en profundidad

- **Representación de los estados: x un número entero.**
- **Número de estados: infinito.**
- **Estado inicial: 0.**
- **Estado final: -3.**
- **Operadores:**
  - **Moverse un paso a la derecha.**
  - **Moverse un paso a la izquierda.**

Problema sin solución en profundidad

- **Implementación del problema del paseo**

```
(defparameter *estado-inicial* 0)

(defparameter *estado-final* -3)

(defun es-estado-final (estado)
  (= estado *estado-final*))

(defparameter *operadores*
  '(mover-a-derecha
    mover-a-izquierda))
```

Problema sin solución en profundidad

```
(defun mover-a-derecha (estado)
  (+ estado 1))

(defun mover-a-izquierda (estado)
  (- estado 1))

(defun aplica (operador estado)
  (funcall (symbol-function operador) estado))
```

Problema sin solución en profundidad

- **Resolución del problema del paseo**

```
> (load "p-paseo.lsp")
T
> (load "b-profundidad.lsp")
T
> (trace es-estado-final)
(ES-ESTADO-FINAL)
> (busqueda-en-profundidad)

1. Trace: (ES-ESTADO-FINAL '0)
1. Trace: (ES-ESTADO-FINAL '1)
1. Trace: (ES-ESTADO-FINAL '2)
1. Trace: (ES-ESTADO-FINAL '3)
*** - PRINT: User break
1. Break> abort
```

Problema sin solución en profundidad

```
> (load "b-anchura.lsp")
T
> (busqueda-en-anchura)
1. Trace: (ES-ESTADO-FINAL '0)
1. Trace: (ES-ESTADO-FINAL '1)
1. Trace: (ES-ESTADO-FINAL '-1)
1. Trace: (ES-ESTADO-FINAL '2)
1. Trace: (ES-ESTADO-FINAL '-2)
1. Trace: (ES-ESTADO-FINAL '3)
1. Trace: (ES-ESTADO-FINAL '-3)
#S(NODO :ESTADO -3
        :CAMINO (MOVER-A-IZQUIERDA MOVER-A-IZQUIERDA MOVER-A-IZQUIERDA))
```

Problema resoluble por profundidad y no por anchura

```
> (load "p-8-puzzle.lsp")
T
> (load "b-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: 0.709987 sec.
Run time: 0.71 sec.
Space: 10660 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))
```

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)
```

Procedimiento de búsqueda en profundidad acotada

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

Procedimiento de búsqueda en profundidad acotada

```
2. Mientras que ABIERTOS no esté vacía,
   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 si la profundidad del ACTUAL es menor que la cota, hacer
             2.4.2.1 NUEVOS-SUCESORES la lista de sucesores del nodo
                     ACTUAL que no están en ABIERTOS ni en CERRADOS y
             2.4.2.2 ABIERTOS la lista obtenida añadiendo los
                     NUEVOS-SUCESORES al principio de ABIERTOS.
3. Si ABIERTOS está vacía, devolver NIL.
```

Implementación de la 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 (append nuevos-sucesores abiertos)))))))
```

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 la cota es mayor que la profundidad de la solución.**
- **No es minimal.**

Comparación de profundidad acotada

- **Solución del 8-puzzle por profundidad acotada:**

```
> (load "p-8-puzzle.lsp")
T
> (load "b-profundidad-acotada.lsp")
T
> (time (busqueda-en-profundidad-acotada))
Real time: 1.212106 sec.
Run time: 1.21 sec.
Space: 17704 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 de 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: 3.242785 sec.
Run time: 3.25 sec.
Space: 45796 Bytes
NIL
```

Comparación de profundidad acotada

```
> (time (busqueda-en-profundidad-acotada :cota 12))
Real time: 0.739477 sec.
Run time: 0.72 sec.
Space: 10756 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))
```

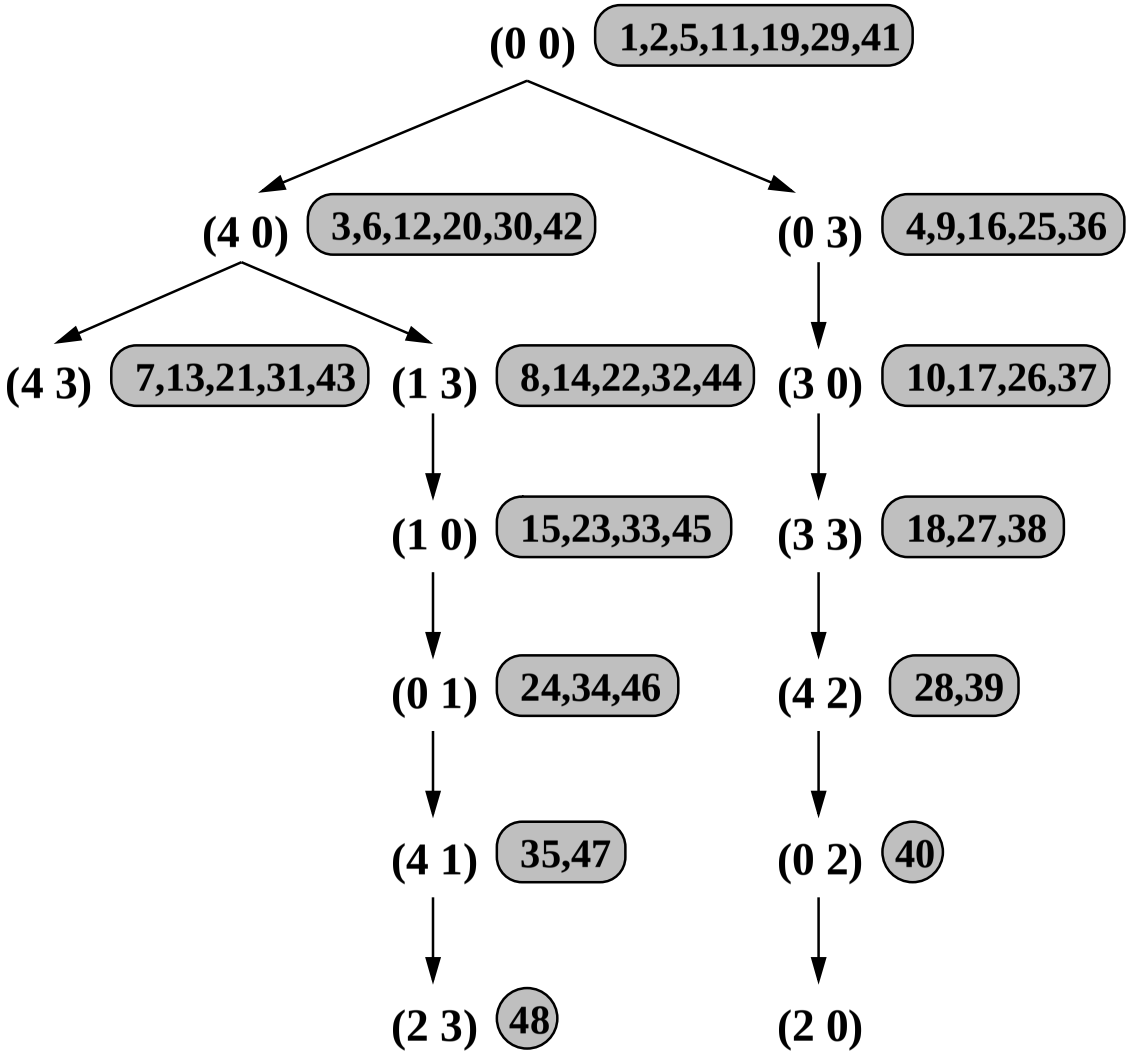
Comparación de profundidad acotada

- **Estadísticas 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)
2 8 3 1 6 4 7 _ 5	4.51	68.292	—	—	1.21	17.704	134.42	940.096
4 8 1 3 _ 2 7 6 5	—	—	0.71	10.660	—	—	0.72	10.756

Problema de las jarras

- **Grafo de búsqueda en profundidad iterativa:**



Implementación de la búsqueda en profundidad iterativa

- **Procedimiento:**
  - **Para buscar la solución por profundidad iterativa se busca por profundidad acotada, partiendo de la cota inicial e incrementándola en uno hasta que se encuentre una solución.**
- **Implementación:**

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

Aplicación al problema de las jarras

```
> (load "p-jarras-1.lsp")
T
> (load "b-profundidad-iterativa.lsp")
T
> (time (busqueda-en-profundidad-iterativa :cota-inicial 0))
Real time: 0.178446 sec.
Run time: 0.18 sec.
Space: 18288 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))
```

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 minimal.**

Comparación de profundidad iterativa

- **Estadísticas de soluciones del 8-puzzle:**

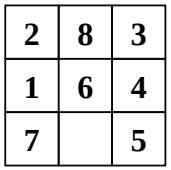
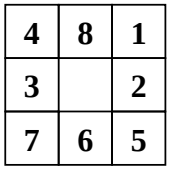
	Anchura		Profundidad		Profundidad iterativa	
Estado inicial	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)
2 8 3 1 6 4 7 _ 5	4.51	68.292	—	—	0.6	8.992
4 8 1 3 _ 2 7 6 5	—	—	0.71	10.660	71.83	655.594

Lisp: Compilación

```
> (compile-file "p-8-puzzle.lsp")
Compiling file p-8-puzzle.lsp ...
Compilation of file p-8-puzzle.lsp is finished.
0 errors, 0 warnings
> (load "p-8-puzzle")
> (compile-file "b-anchura.lsp")
> (load "b-anchura")
> (time (busqueda-en-anchura))
Real time: 0.21704 sec.
Run time: 0.22 sec.
Space: 27388 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 de profundidad iterativa compilada

- **Estadísticas del 8-puzzle con procedimientos compilados:**

	Anchura		Profundidad		Profundidad iterativa	
Estado inicial	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)
	0.22	27.424	—	—	0.06	8.236
	—	—	0.04	5.984	15.88	656.020

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 $c >= p$	Sí
Minimal	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".**
- **[Cortés–94] Cap. 4: "Búsqueda y estrategias".**
- **[Fernández, Gónzalez y Mira, 1998] Cap. 1 "Búsqueda sin información del dominio"**
- **[Mira–95] Cap. 3: "Fundamentos y técnicas básicas de búsqueda".**
- **[Rich–94] Cap. 2 "Problemas, espacios problema y búsqueda".**
- **[Russell y Norvig, 1995] Cap. 3 "Solving problems by search"**
- **[Winston–91] Cap. 19 "Ejemplos que involucran búsquedas".**
- **[Winston-94] Cap. 4: "Redes y búsqueda básica".**