Tema 4: Técnicas heurísticas de búsqueda

José A. Alonso Jiménez
Francisco J. Martín Mateos

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Concepto de heurística

- **Objetivo de la búsqueda heurística: Podar el espacio de búsqueda.**
- **Base de la heurística: Comparación de los estados.**
- **Función de evaluación heurística:**
  - **Estima la distancia al final.**
  - **Valor en el estado final: minimal.**
- **Comparación de los estados mediante valor heurístico.**

Problema del paseo

- **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.**

Heurística en el problema del paseo

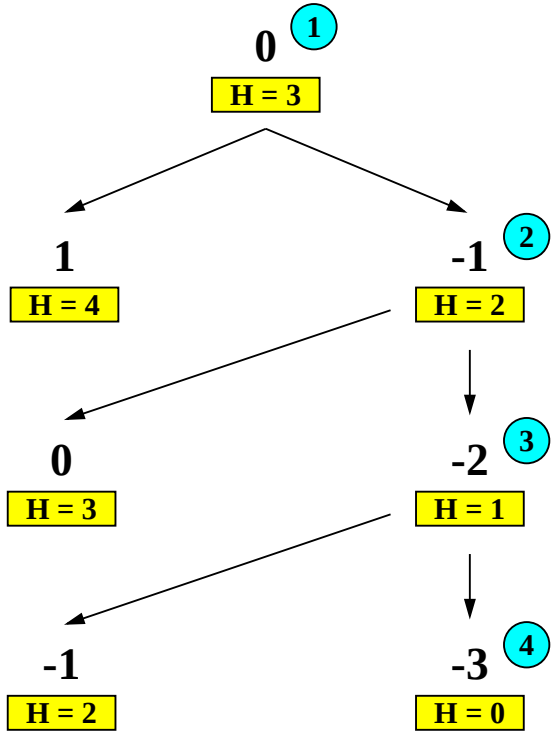
- **Función de evaluación heurística:**

```
heuristica(estado) = distancia(estado,estado-final)
```

- **Representación:**

```
(defun heuristica (estado)
  (abs (- estado *estado-final*)))
```

Grafo de escalada para el problema del paseo



Definición de nodo heurístico

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

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

Procedimiento de búsqueda en escalada

1. Crear la variable local ACTUAL que es el nodo heurístico cuyo estado es el *ESTADO-INICIAL*, cuyo camino es la lista vacía y cuya heurística es la del *ESTADO-INICIAL*.
2. Repetir mientras que el nodo ACTUAL no sea nulo:
   2.1. si el estado del nodo ACTUAL es un estado final, devolver el nodo ACTUAL y terminar;
   2.2. en caso contrario, cambiar ACTUAL por su mejor sucesor (es decir, uno de sus sucesores cuya heurística sea menor que la del nodo ACTUAL y menor o igual que las heurísticas de los restantes sucesores, si existen dichos sucesores y NIL en caso contrario).

Implementación de la búsqueda en escalada

```
(defun busqueda-en-escalada ()
  (let ((actual (crea-nodo-h :estado *estado-inicial*          ; 1
                             :camino nil
                             :heuristica-del-nodo
                             (heuristica *estado-inicial*))))
    (loop until (null actual) do                               ; 2
          (cond ((es-estado-final (estado actual))             ; 2.1
                 (return actual))
                (t (setf actual                                ; 2.2
                         (mejor (sucesores actual)
                                (heuristica-del-nodo actual)))))))))
```

Implementación de la búsqueda en escalada

```
(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-h :estado siguiente-estado
                       :camino (cons operador (camino nodo))
                       :heuristica-del-nodo
                       (heuristica siguiente-estado)))))
```

Implementación de la búsqueda en escalada

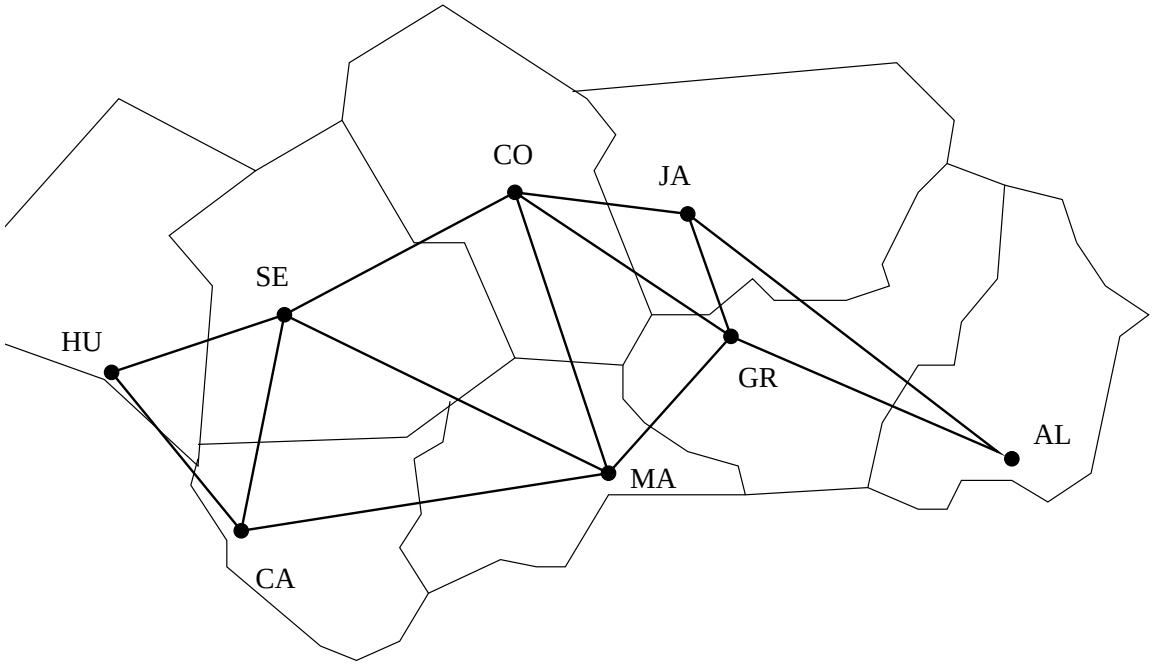
```
(defun mejor (nodos minima-distancia-al-final)
  (let ((mejor-nodo nil))
    (loop for n in nodos do
      (when (< (heuristica-del-nodo n) minima-distancia-al-final)
            (setf mejor-nodo n
                  minima-distancia-al-final (heuristica-del-nodo n))))
    mejor-nodo))
```

Solución del paseo en escalada

```
> (load "p-paseo.lsp")
T
> (load "b-escalada.lsp")
T
> (trace es-estado-final)
(ES-ESTADO-FINAL)
> (busqueda-en-escalada)
1. Trace: (ES-ESTADO-FINAL '0)
1. Trace: (ES-ESTADO-FINAL '-1)
1. Trace: (ES-ESTADO-FINAL '-2)
1. Trace: (ES-ESTADO-FINAL '-3)
#S(NODO-H
   :ESTADO -3
   :CAMINO (MOVER-A-IZQUIERDA MOVER-A-IZQUIERDA MOVER-A-IZQUIERDA)
   :HEURISTICA-DEL-NODO 0)
```

Heurística en el problema del viaje

- **Mapa:**



Heurística en el problema del viaje

- **Coordenadas:**

```
Almeria : (409.5  93  )
Granada : (309   127.5)
Malaga  : (232.5  75  )
Cadiz   : ( 63    57  )
Huelva  : (  3   139.5)
Sevilla : ( 90   153  )
Cordoba : (198   207  )
Jaen    : (295.5 192  )
```

- **Función de evaluación heurística:**

```
heuristica(estado) = distancia(coordenadas(estado),
                               coordenadas(almeria))
```

Lisp: Funciones matemáticas y listas de asociación

- **Funciones matemáticas**

```
* (SQRT X)
        (sqrt 144)  =>  12

* (EXPT X Y)
        (expt 2 4)  =>  16
```

- **Listas de asociación**

```
* (ASSOC ITEM A-LISTA [:TEST PREDICADO])
        (assoc 'b '((a 1) (b 2) (c 3)))                    =>  (B 2)
        (assoc '(b) '((a 1) ((b) 1) (c d)))                =>  NIL
        (assoc '(b) '((a 1) ((b) 1) (c d)) :test #'equal)  =>  ((B) 1)
```

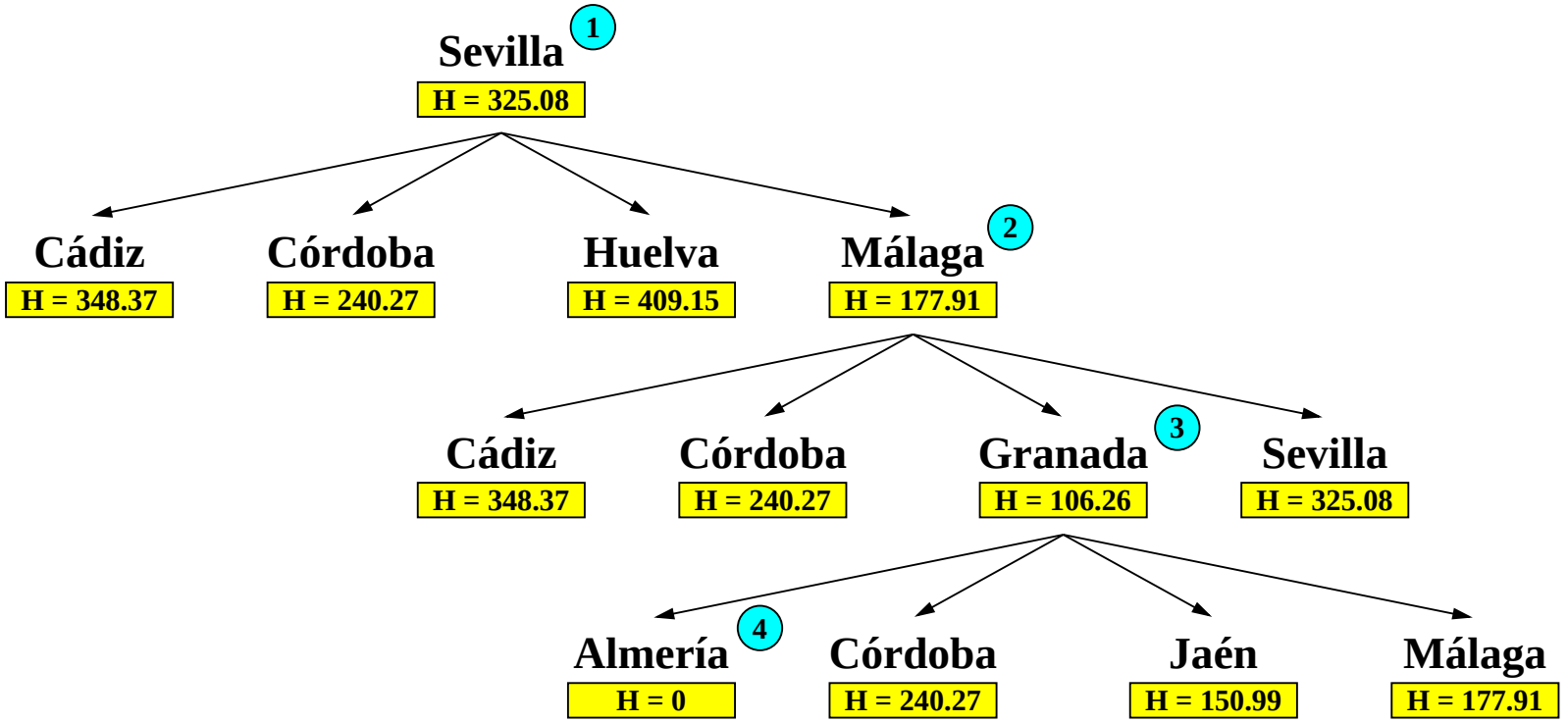
Implementación del problema de viaje con heurística

```
(defun heuristica (estado)
  (distancia estado *estado-final*))

(defun distancia (c1 c2)
  (sqrt (+ (expt (- (abscisa c1) (abscisa c2)) 2)
           (expt (- (ordenada c1) (ordenada c2)) 2))))

(defun abscisa (ciudad)
  (first (second (assoc ciudad *ciudades*))))

(defun ordenada (ciudad)
  (second (second (assoc ciudad *ciudades*))))
```

Implementación del problema de viaje con heurística

```
(defparameter *ciudades*
  '((almeria (409.5  93))
    (cadiz   ( 63    57))
    (cordoba (198   207))
    (granada (309   127.5))
    (huelva  (  3   139.5))
    (jaen    (295.5 192))
    (malaga  (232.5  75))
    (sevilla ( 90   153))))
```

Grafo de escalada para el problema del viaje



Solución del viaje en escalada

```
> (load "p-viaje.lsp")
T
> (load "b-escalada.lsp")
T
> (trace es-estado-final)
(ES-ESTADO-FINAL)
> (busqueda-en-escalada)
1. Trace: (ES-ESTADO-FINAL 'SEVILLA)
1. Trace: (ES-ESTADO-FINAL 'MALAGA)
1. Trace: (ES-ESTADO-FINAL 'GRANADA)
1. Trace: (ES-ESTADO-FINAL 'ALMERIA)
#S(NODO-H
   :ESTADO ALMERIA
   :CAMINO (IR-A-ALMERIA IR-A-GRANADA IR-A-MALAGA)
   :HEURISTICA-DEL-NODO 0.0)
```

Primera heurística en el problema del 8–puzzle

- **Heurística: Número de piezas descolocadas.**

2	8	3
1	6	4
7		5

H = 4

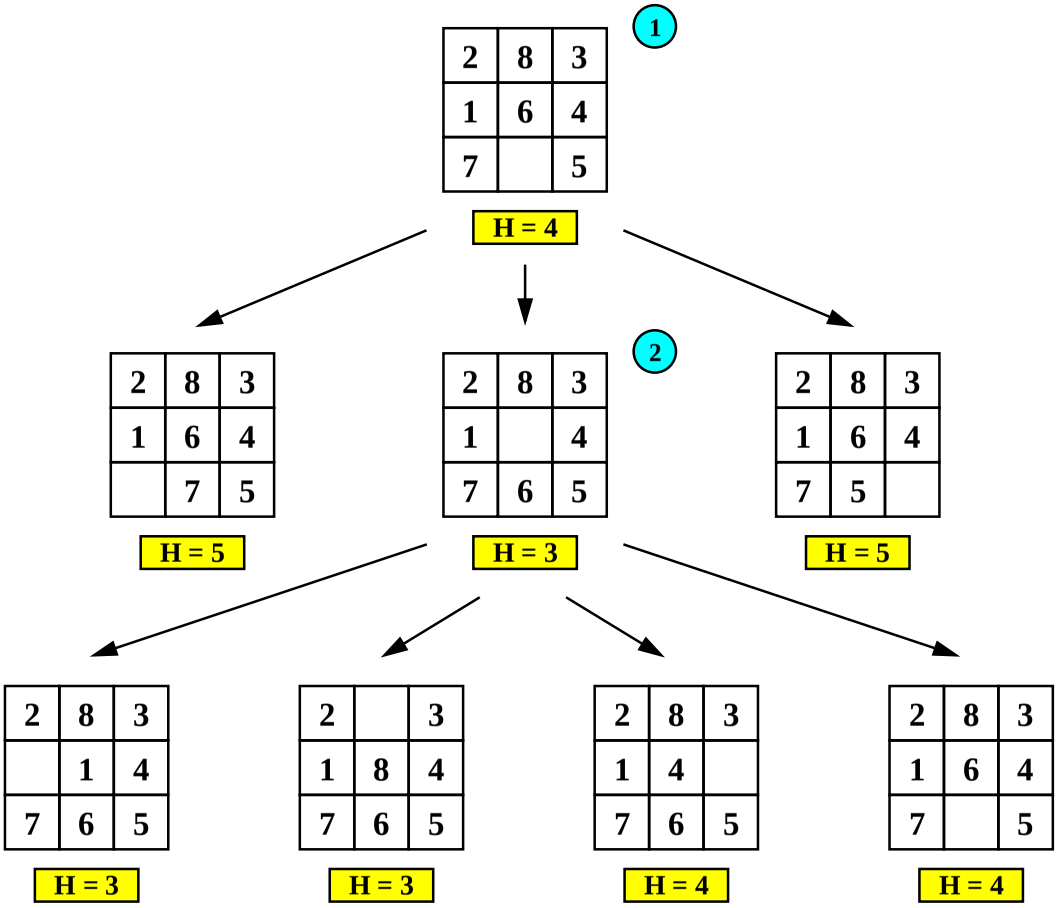
1	2	3
8		4
7	6	5

H = 0

- **Representación:**

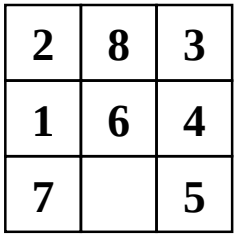
```
(defun heuristica (estado)
  (loop for i from 1 to 8
    counting (not (equal (coordenadas i estado)
                         (coordenadas i *estado-final*)))))
```

8–puzzle por escalada: 1ª heurística

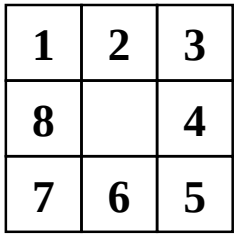


Segunda heurística en el problema del 8–puzzle

- **Heurística: Suma de las distancias manhattan.**



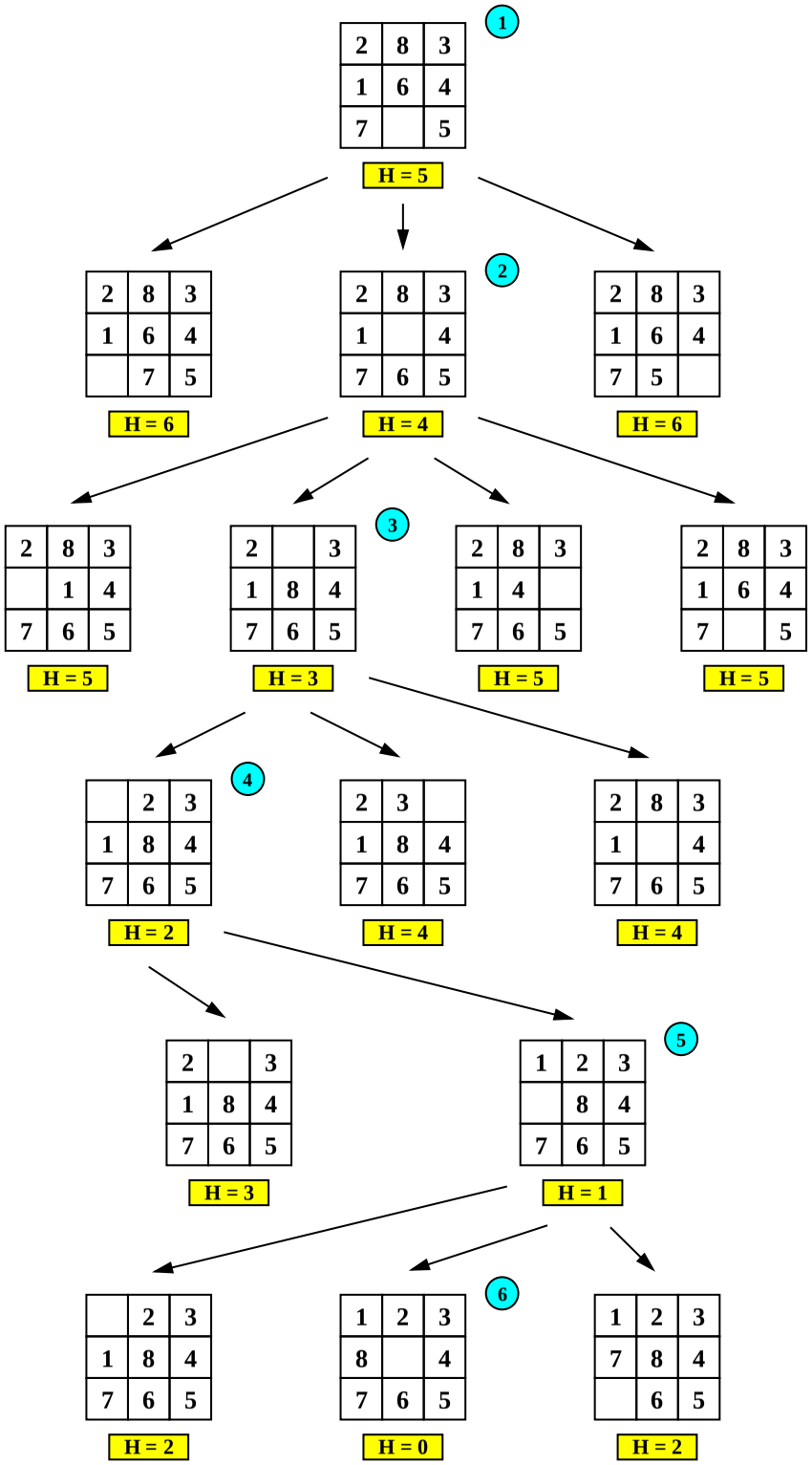
H = 5



H = 0

```
(defun heuristica-2 (estado)
  (loop for i from 1 to 8
        summing (distancia-manhattan (coordenadas i estado)
                                     (coordenadas i *estado-final*))))

(defun distancia-manhattan (c1 c2)
  (+ (abs (- (first c1) (first c2)))
     (abs (- (second c1) (second c2)))))
```

8–puzzle por escalada: 2ª heurística



Comparación de escalada en el 8–puzzle

	Profundidad iterativa		Escalada	
Estado inicial	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)
2 8 3 1 6 4 7 _ 5	0.06	9.028	0.05	6.516
4 8 1 3 _ 2 7 6 5	15.88	656.020	0.11	14.604

Propiedades de la búsqueda por escalada

- **Complejidad:**
  - **p: profundidad de la solución.**
  - **Complejidad en espacio: $O(1)$.**
  - **Complejidad en tiempo: $O(p)$.**
- **No es completa.**
- **No es minimal.**
- **Problemas:**
  - **Máximos locales.**
  - **Mesetas.**

Lisp: Ordenación

```
* (SORT LISTA PREDICADO [:KEY CLAVE])
      (sort '(3 1 5 2) #'<)                    =>  (1 2 3 5)
      (sort '(-3 1 -5 2 7) #'>)                =>  (7 2 1 -3 -5)
      (sort '(-3 1 -5 2 7) #'> :key #'abs)     =>  (7 -5 -3 2 1)
      (setf l '(a c b d))                      =>  (A C B D)
      (sort l #'string<)                       =>  (A B C D)
```

Procedimiento de la búsqueda por primero el mejor

1. Crear las siguientes variables locales
   1.1. ABIERTOS (para almacenar los nodos generados aún no analizados) con valor la lista formado por el nodo inicial (es decir, el nodo cuyo estado es el estado inicial, cuyo camino es la lista vacía y cuya heurística es la del estado inicial);
   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 la búsqueda por primero el mejor

```
2. Mientras que ABIERTOS no está 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 y ordenando sus
                     nodos por orden creciente de sus heurísticas.
3. Si ABIERTOS está vacía, devolver NIL.
```

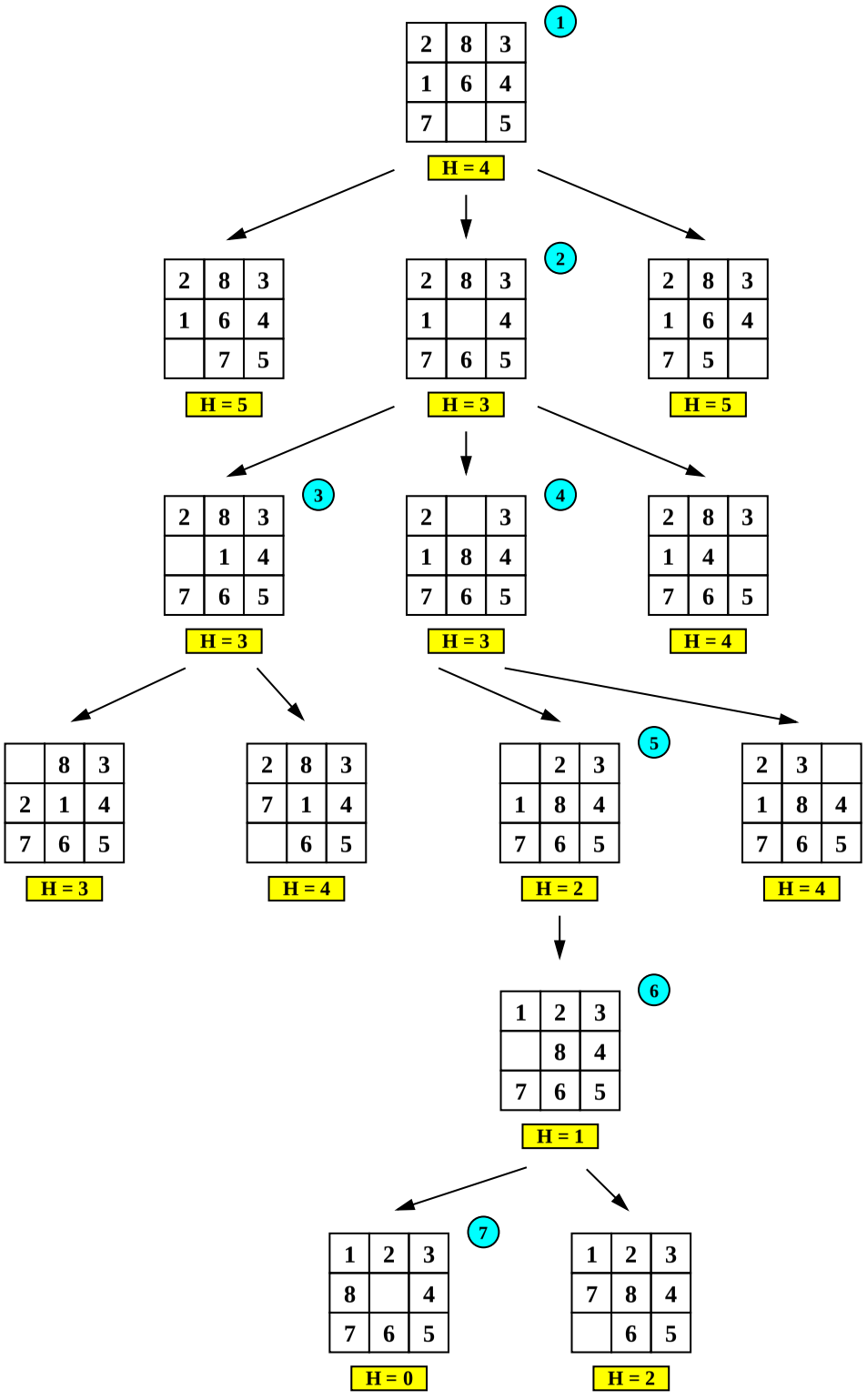
Implementación de la búsqueda por primero el mejor

```
(defun busqueda-por-primero-el-mejor ()
  (let ((abiertos (list (crea-nodo-h :estado *estado-inicial*
                                     :camino nil
                                     :heuristica-del-nodo
                                     (heuristica *estado-inicial*)))) ;1.1
        (cerrados nil)                                                ;1.2
        (actual nil)                                                  ;1.3
        (nuevos-sucesores nil))                                       ;1.4
```

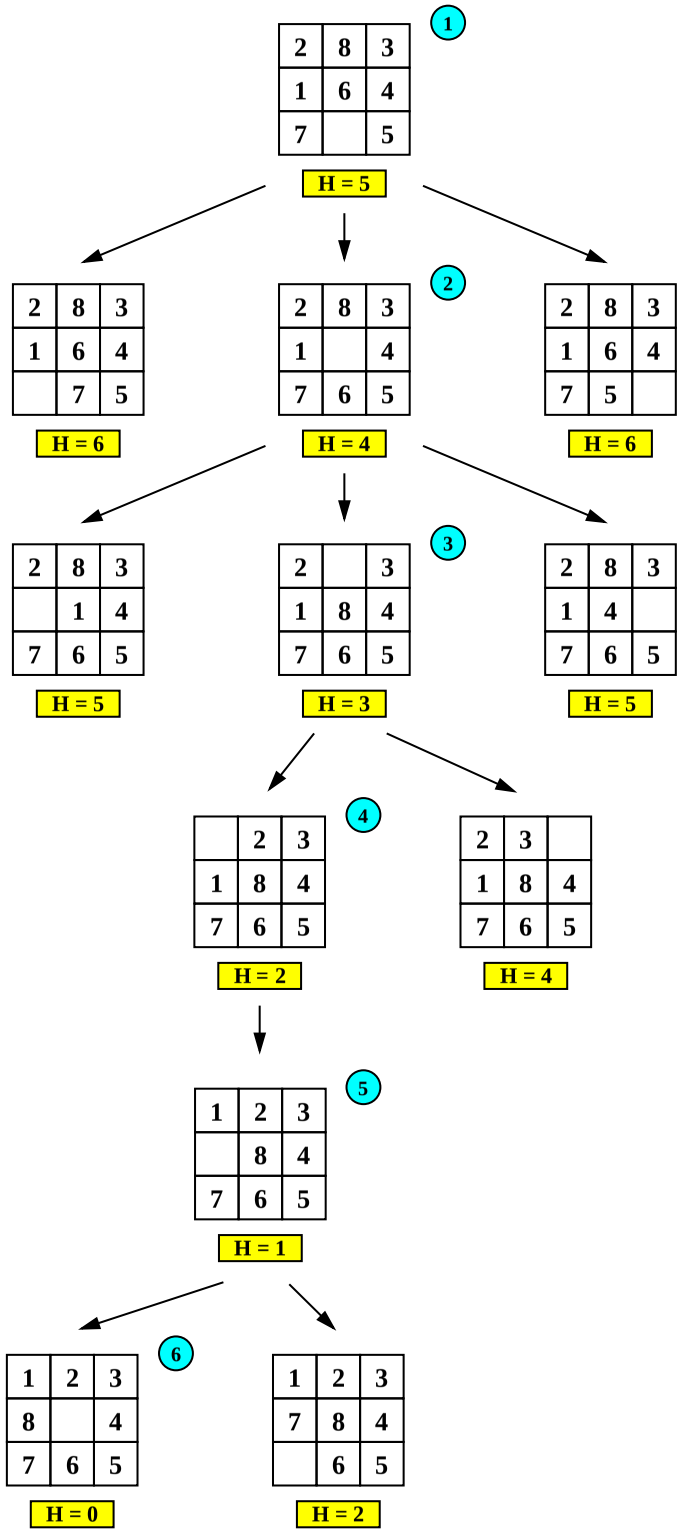
Implementación de la búsqueda por primero el mejor

```
(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
                 (ordena-por-heuristica
                  (append abiertos nuevos-sucesores)))))))

(defun ordena-por-heuristica (lista-de-nodos)
  (sort lista-de-nodos #'< :key #'heuristica-del-nodo))
```

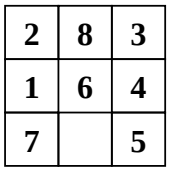
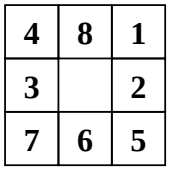
8–puzzle por primero el mejor: 1ª heurística



8–puzzle por primero-el-mejor: 2ª heurística



Comparación de primero el mejor en el 8–puzzle

	Profundidad iterativa		Escalada		Primero el mejor	
Estado inicial	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)	Tiempo (seg.)	Espacio (bytes)
	0.06	9.028	0.05	6.516	0.05	6.884
	15.88	656.020	0.11	14.604	0.13	15.844

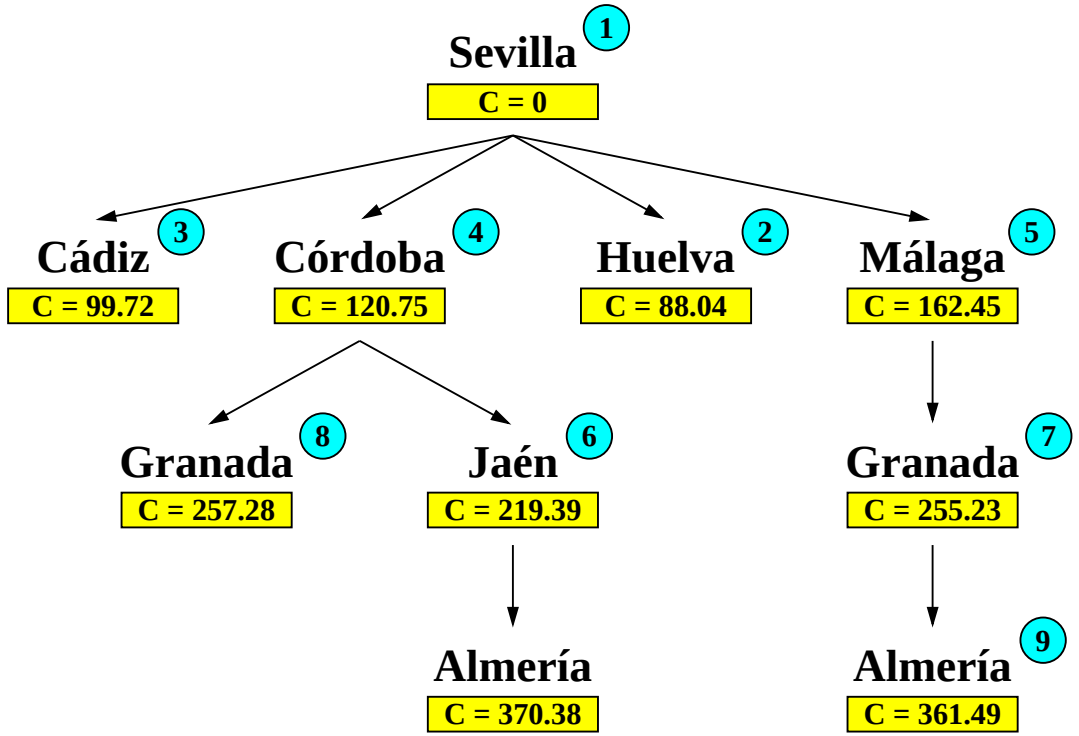
Propiedades de la búsqueda por primero el mejor

- **Complejidad:**
  - r: **factor de ramificación.**
  - p: **profundidad de la solución.**
  - **Complejidad en espacio:** $O(r^p)$.
  - **Complejidad en tiempo:** $O(r^p)$.
- **Es completa.**
- **No es minimal.**

Costes en el problema del viaje

```
(defun coste-de-aplicar-operador (estado operador)
  (let ((estado-sucesor (aplica operador estado)))
    (when estado-sucesor
          (distancia estado estado-sucesor))))
```

Grafo optimal para el viaje



Definición de nodo de coste

- **Nodo de coste = Estado + Camino + Coste del camino**
- **Representación de nodos en Lisp**

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

Procedimiento de búsqueda optimal

1. Crear las siguientes variables locales
   1.1. ABIERTOS (para almacenar los nodos de coste generados aún no analizados) con valor la lista formado por el nodo inicial (es decir, el nodo cuyo estado es el estado inicial, cuyo camino es la lista vacía y cuyo coste es 0);
   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. SUCESORES (para almacenar la lista de los sucesores del nodo actual) con valor la lista vacía.

Procedimiento de búsqueda optimal

```
2. Mientras que ABIERTOS no está 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 SUCESORES la lista de sucesores del nodo ACTUAL para
                     los que no existen en ABIERTOS o CERRADOS un nodo con
                     el mismo estado y menor coste.
             2.4.2.2 ABIERTOS la lista obtenida añadiendo los NUEVOS-SUCESORES
                     al final de ABIERTOS y ordenando sus nodos por orden
                     creciente de sus costes.
3. Si ABIERTOS está vacía, devolver NIL.
```

Implementación de la búsqueda optimal

```
(defun busqueda-optimal ()
  (let ((abiertos (list (crea-nodo-c :estado *estado-inicial*    ;1.1
                                     :camino nil
                                     :coste-camino 0)))
        (cerrados nil)                                           ;1.2
        (actual nil)                                             ;1.3
        (sucesores nil))                                         ;1.4
```

Implementación de la búsqueda optimal

```
(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 sucesores                                              ;2.4.2.1
                 (nuevos-o-mejores-sucesores actual abiertos cerrados))
           (setf abiertos                                               ;2.4.2.2
                 (ordena-por-coste (append abiertos sucesores)))))))
```

Implementación de la búsqueda optimal

```
(defun nuevos-o-mejores-sucesores (nodo abiertos cerrados)
  (elimina-peores (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)))
```

Implementación de la búsqueda optimal

```
(defun sucesor (nodo operador)
  (let ((siguiente-estado (aplica operador (estado nodo))))
    (when siguiente-estado
          (crea-nodo-c :estado siguiente-estado
                       :camino (cons operador (camino nodo))
                       :coste-camino
                       (+ (coste-de-aplicar-operador (estado nodo)
                                                     operador)
                          (coste-camino nodo))))))

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

Implementación de la búsqueda optimal

```
(defun esta-mejor (nodo lista-de-nodos)
  (let ((estado (estado nodo)))
    (loop for n in lista-de-nodos
      thereis (and (equalp estado (estado n))
                   (<= (coste-camino n) (coste-camino nodo))))))

(defun ordena-por-coste (lista-de-nodos)
  (sort lista-de-nodos #'< :key #'coste-camino))
```

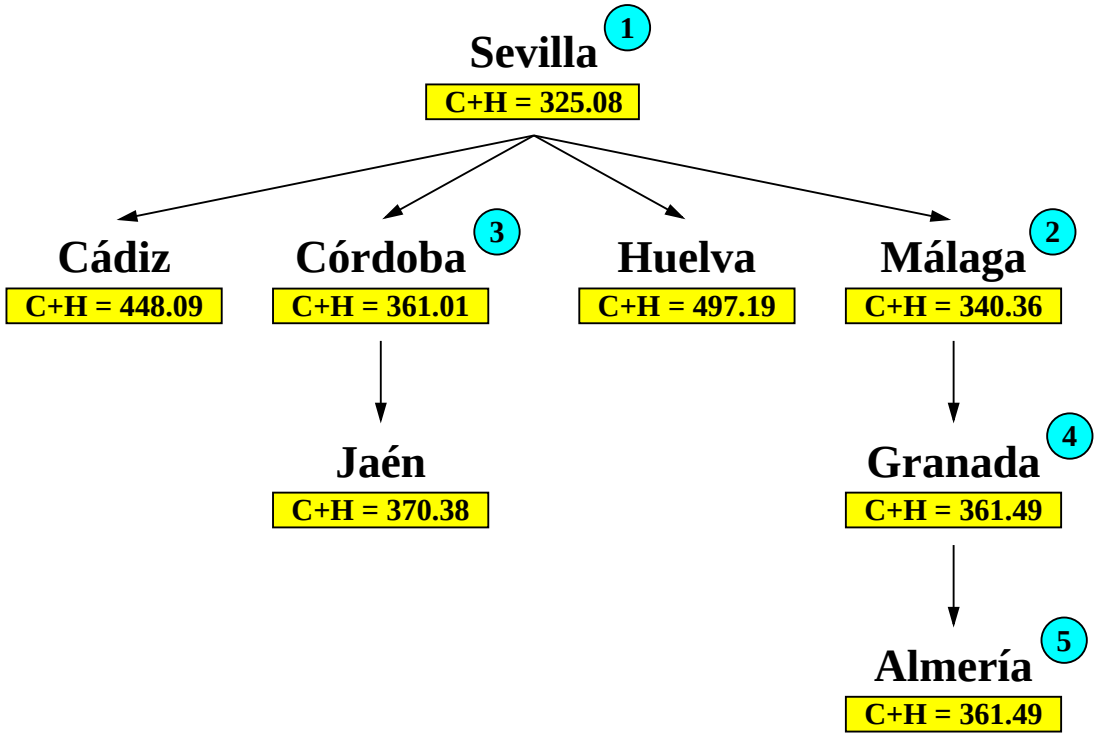
El viaje mediante búsqueda optimal

```
> (load "p-viaje")
T
> (load "b-optimal")
T
> (busqueda-optimal-con-traza)
SEVILLA Coste: 0.00
  CADIZ Coste: 99.72
  CORDOBA Coste: 120.75
  HUELVA Coste: 88.04
  MALAGA Coste: 162.45
HUELVA Coste: 88.04
CADIZ Coste: 99.72
CORDOBA Coste: 120.75
  GRANADA Coste: 257.28
  JAEN Coste: 219.39
MALAGA Coste: 162.45
  GRANADA Coste: 255.23
JAEN Coste: 219.39
  ALMERIA Coste: 370.38
GRANADA Coste: 255.23
  ALMERIA Coste: 361.49
GRANADA Coste: 257.28
#S(NODO-C :ESTADO ALMERIA
          :CAMINO (IR-A-ALMERIA IR-A-GRANADA
                                IR-A-MALAGA)
          :COSTE-CAMINO 361.48953)
```

Propiedades de la búsqueda optimal

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

Grafo A* para el viaje



Definición de nodo de coste con heurística

- **Nodo de coste = Estado + Camino + Coste-camino + Coste-más-heurística**
- **Representación de nodos en Lisp**

```
(defstruct (nodo-hc (:constructor crea-nodo-hc)
                    (:conc-name nil))
  estado
  camino
  coste-camino
  coste-mas-heuristica)
```

Procedimiento A*

1. Crear las siguientes variables locales
   1.1. ABIERTOS (para almacenar los nodos heurísticos con costes generados aún no analizados) con valor la lista formado por el nodo inicial (es decir, el nodo cuyo estado es el estado inicial, cuyo camino es la lista vacía, el coste es 0 y el coste más heurística es la heurística del estado inicial);
   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. SUCESORES (para almacenar la lista de los sucesores del nodo actual) con valor la lista vacía.

Procedimiento A*

```
2. Mientras que ABIERTOS no está 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 SUCESORES la lista de sucesores del nodo ACTUAL para
                     los que no existen en ABIERTOS o CERRADOS un nodo con
                     el mismo estado y menor coste.
             2.4.2.2 ABIERTOS la lista obtenida añadiendo los NUEVOS-SUCESORES
                     al final de ABIERTOS y ordenando sus nodos por orden
                     creciente de sus sumas de costes y heurísticas.
3. Si ABIERTOS está vacía, devolver NIL.
```

Implementación de A*

```
(defun busqueda-a-estrella-1 ()
  (let ((abiertos (list (crea-nodo-hc :estado *estado-inicial*      ;1.1
                                      :camino nil
                                      :coste-camino 0
                                      :coste-mas-heuristica
                                      (heuristica *estado-inicial*))))
        (cerrados nil)                                              ;1.2
        (actual nil)                                                ;1.3
        (sucesores nil))                                            ;1.4
```

Implementación de A*

```
(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 sucesores                                          ;2.4.2.1
                     (nuevos-o-mejores-sucesores actual abiertos cerrados))
               (setf abiertos                                           ;2.4.2.2
                     (ordena-por-coste-mas-heuristica
                       (append abiertos sucesores)))))))
```

Implementación de A*

```
(defun sucesor (nodo operador)
  (let ((siguiente-estado (aplica operador (estado nodo))))
    (when siguiente-estado
      (crea-nodo-hc :estado siguiente-estado
                    :camino (cons operador (camino nodo))
                    :coste-camino
                    (+ (coste-de-aplicar-operador (estado nodo) operador)
                       (coste-camino nodo))
                    :coste-mas-heuristica
                    (+ (+ (coste-de-aplicar-operador (estado nodo) operador)
                          (coste-camino nodo))
                       (heuristica siguiente-estado))))))

(defun ordena-por-coste-mas-heuristica (lista-de-nodos)
  (sort lista-de-nodos #'< :key #'coste-mas-heuristica))
```

El viaje mediante A*

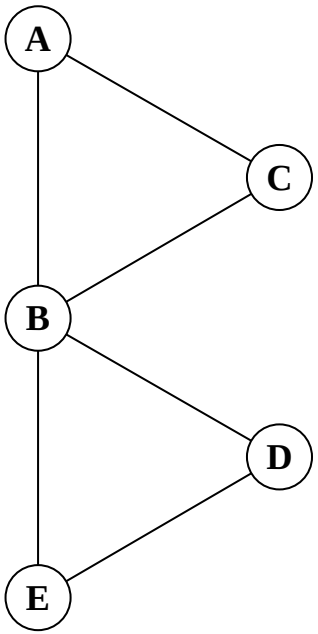
```
> (load "p-viaje")
T
> (load "b-a-estrella-1.lsp")
T
> (busqueda-a-estrella-1-con-traza)
SEVILLA Coste+Heuristica: 325.08
  CADIZ Coste+Heuristica: 448.09
  CORDOBA Coste+Heuristica: 361.01
  HUELVA Coste+Heuristica: 497.19
  MALAGA Coste+Heuristica: 340.36
MALAGA Coste+Heuristica: 340.36
  GRANADA Coste+Heuristica: 361.49
CORDOBA Coste+Heuristica: 361.01
  JAEN Coste+Heuristica: 370.38
GRANADA Coste+Heuristica: 361.49
  ALMERIA Coste+Heuristica: 361.49
#S(NODO-HC
   :ESTADO ALMERIA
   :CAMINO (IR-A-ALMERIA IR-A-GRANADA IR-A-MALAGA)
   :COSTE-CAMINO 361.48953
   :COSTE-MAS-HEURISTICA 361.48953))
```

Propiedades de A*

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

Problema del grafo

- **Enunciado:**
  - **En el siguiente grafo buscar el camino de menor coste desde A hasta E.**



Costes

	A	B	C	D	E
A		8	1		
B	8		2	3	7
C	1	2			
D		3			1
E		7		1	

Heurísticas

A	B	C	D	E
5	1	10	3	0

Representación del problema del grafo

```
(defparameter *estado-inicial* 'a)

(defun es-estado-final (estado)
  (eq estado 'e))

(defparameter *operadores*
  '(ir-a-b
    ir-a-c
    ir-a-d
    ir-a-e))

(defun ir-a-b (estado)
  (when (member estado '(a c d e))
        'b))

(defun ir-a-c (estado)
  (when (member estado '(a b))
        'c))

(defun ir-a-d (estado)
  (when (member estado '(b e))
        'd))

(defun ir-a-e (estado)
  (when (member estado '(b d))
        'e))
```

Representación del problema del grafo

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

(defun heuristica (estado)
  (case estado
        (a 5)
        (b 1)
        (c 10)
        (d 3)
        (e 0)))

(defun coste-de-aplicar-operador (estado operador)
  (case estado
        (a (case operador
                 (ir-a-b 8)
                 (ir-a-c 1)))
        (b (case operador
                 (ir-a-c 2)
                 (ir-a-d 3)
                 (ir-a-e 7)))
        (c 2)
        (d (case operador
                 (ir-a-b 3)
                 (ir-a-e 1)))
        (e (case operador
                 (ir-a-b 7)
                 (ir-a-d 1)))))
```

Comparación de soluciones de los misioneros

Problema con 10 misioneros, 10 caníbales y capacidad 6			
Procedimiento	**Coste**	**Segundos**	**Bytes**
en-anchura	28	0.69	170,120
en-profundidad	100	0.13	24,712
en-profundidad-iterativa	76	4.95	886,104
optimal	26	0.93	197,476
H=1 en-escalada	Sin sol		
H=1 por-primero-el-mejor	26	0.21	42,044
H=1 a-estrella	26	2.20	292,316
H=2 en-escalada	Sin sol		
H=2 por-primero-el-mejor	26	0.21	42,044
H=2 a-estrella	26	0.28	48,432

Comparación de soluciones de los misioneros

Problema con 20 misioneros, 20 caníbales y capacidad 11			
Procedimiento	**Coste**	**Segundos**	**Bytes**
en-anchura	48	9.34	1,499,192
en-profundidad	416	3.28	291,388
en-profundidad-iterativa	114	46.68	4,882,516
optimal	46	13.09	1,832,564
H=1 en-escalada	Sin sol		
H=1 por-primero-el-mejor	46	1.36	188,804
H=1 a-estrella	46	43.48	3,090,536
H=2 en-escalada	Sin sol		
H=2 por-primero-el-mejor	46	1.42	188,804
H=2 a-estrella	46	1.90	220,916

Comparación de soluciones de las N reinas

Problema con 8 reinas			
Procedimiento	**Coste**	**Segundos**	**Bytes**
en-anchura	8	18.50	6,605,436
en-profundidad	8	0.10	8,380
en-profundidad-iterativa	8	30.15	329,312
optimal	Agotado		
en-escalada	8	0.01	1,128
por-primero-el-mejor	8	0.02	2,560
a-estrella	8	0.01	2,656

Comparación de soluciones de las N reinas

Problema con 16 reinas			
Procedimiento	**Coste**	**Segundos**	**Bytes**
en-anchura	Agotado		
en-profundidad	16	351.98	687,208
en-profundidad-iterativa	Agotado		
optimal	Agotado		
en-escalada	Sin sol		
por-primero-el-mejor	16	0.51	58,860
a-estrella	16	0.50	59,404

Bibliografía

- **[Borrajo-93] Cap. 5: "Técnicas para restringir la explosión combinatoria".**
- **[Cortés-94]**
  Cap. 4.5: "Búsqueda con información heurística" y
  Cap. 4.6: "Búsqueda de la solución óptima".
- **[Fernández, Gónzalez y Mira, 1998] Cap. 2 "Búsqueda heurística"**
- **[Mira-95] Cap. 4: "Búsqueda heurística".**
- **[Rich-91] Cap. 3: "Técnicas de búsqueda heurística".**
- **[Russell y Norvig, 1995] Cap. 4 "Informed search methods"**
- **[Winston-94]**
  Cap. 4: "Redes y búsqueda básica" y
  Cap. 5: "Redes y búsqueda óptima".