Técnicas heurísticas para la resolución de problemas

José A. Alonso y Francisco J. Martín

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: 0.
- 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

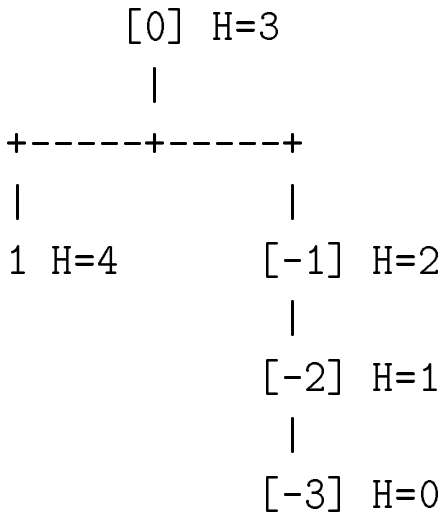
- Función de evaluación heurística:

```
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

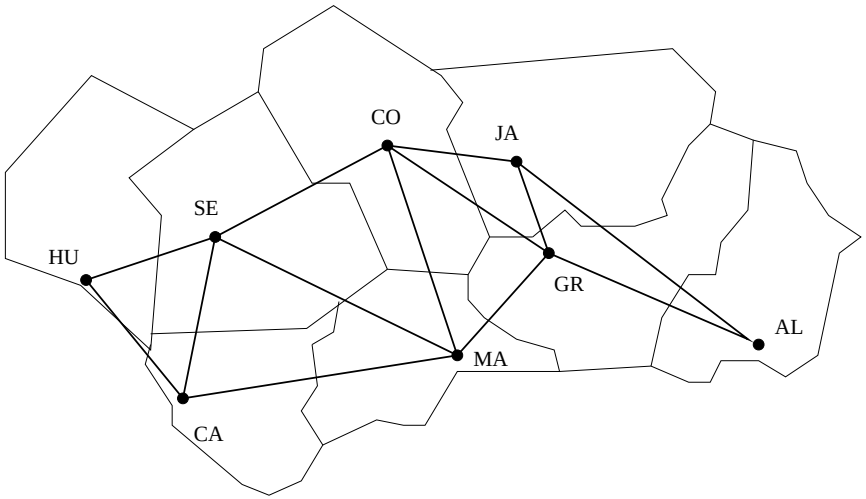
```
(defun mejor (nodos minima-distancia-al-final)
  (let ((mejor-nodo nil))
    (loop for n in nodos do
      (when (< (heuristica-del-nodo n) minima-distancia-al-final)
            (setf mejor-nodo n
                  minima-distancia-al-final (heuristica-del-nodo n))))
    mejor-nodo))
```

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)   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))
```

- 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)
```

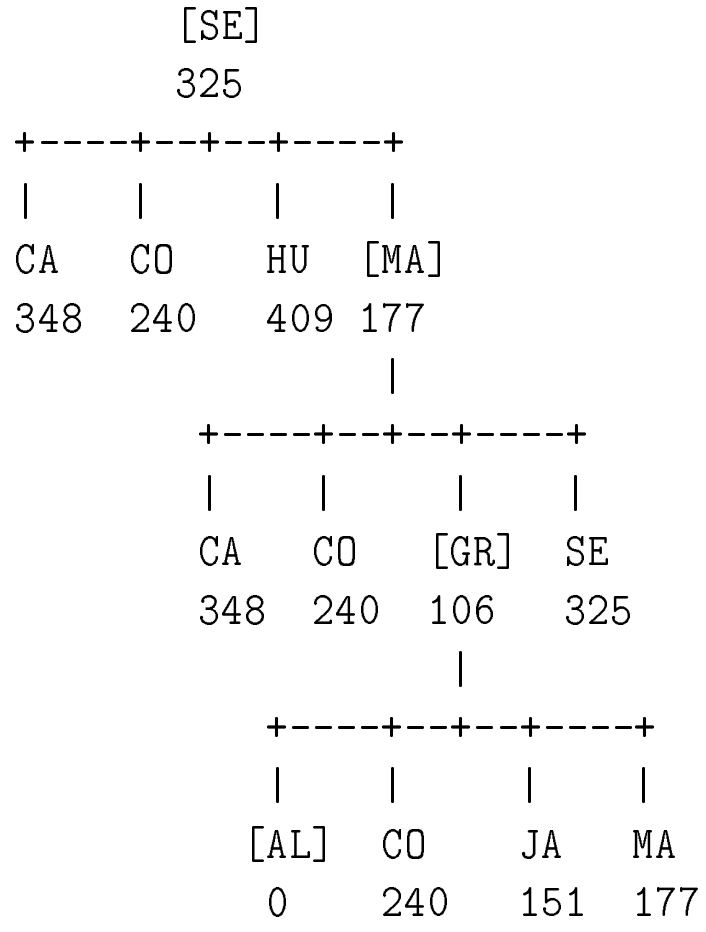
Implementación del problema de viaje con heurística

```
(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.

```
+---+     +---+
|283|     |123|
|164|     |8 4|
|7 5|     |765|
+---+     +---+
 H=4       H=0
```

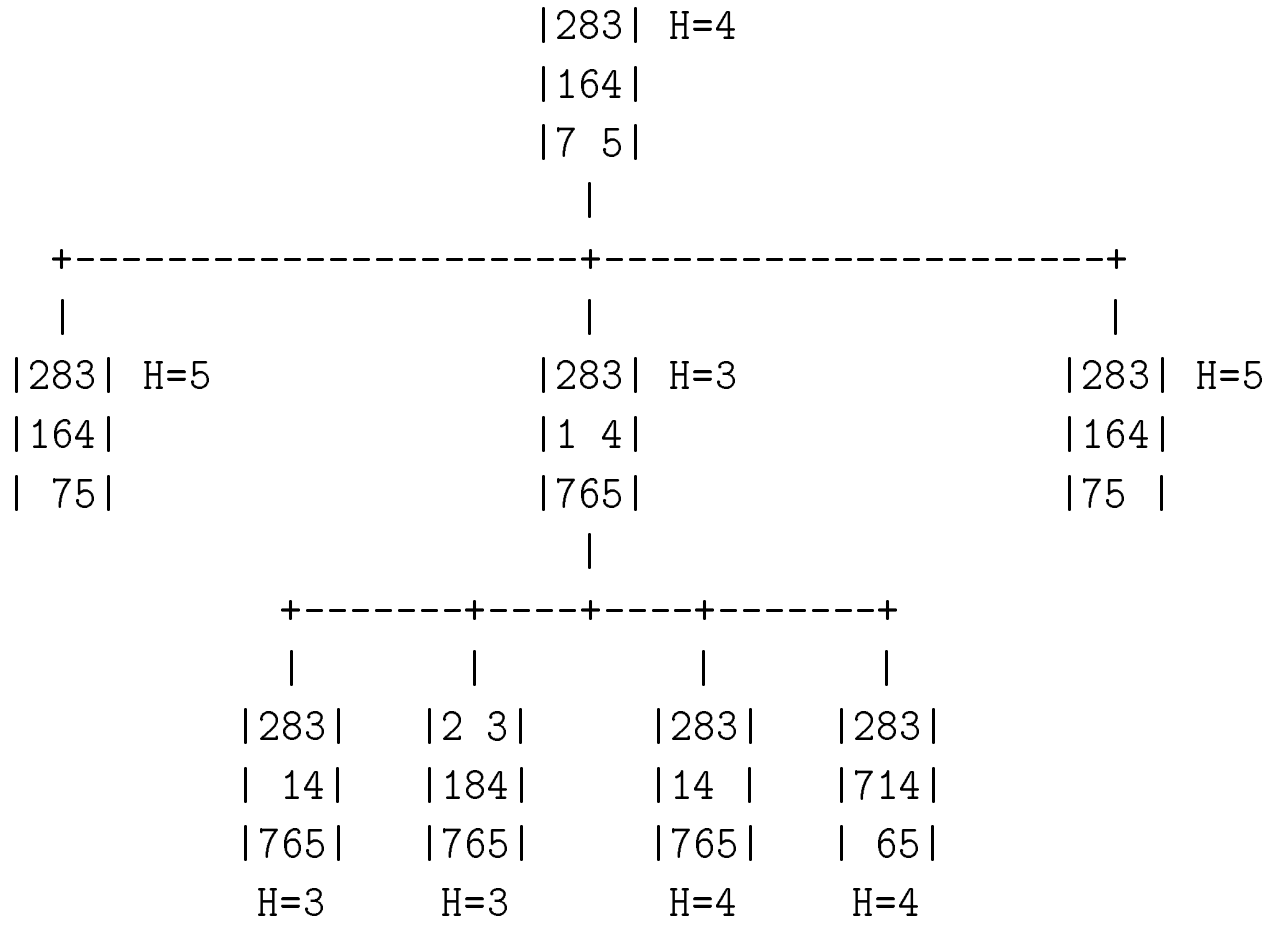
- Representación:

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

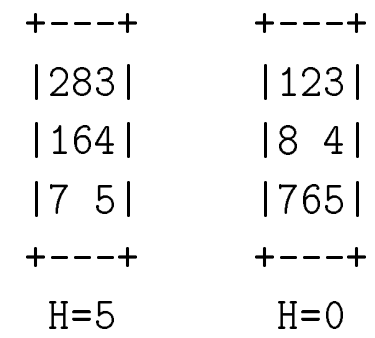
8–puzzle por escalada con la primera heurística

```
> (load "p-8-puzzle")
T
> (load "b-escalada")
T
> (busqueda-en-escalada-con-traza :traza 2)
1: #2A((2 8 3) (1 6 4) (7 H 5)) (H=4.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=5.00)
--> #2A((2 8 3) (1 H 4) (7 6 5)) (H=3.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=5.00)
2: #2A((2 8 3) (1 H 4) (7 6 5)) (H=3.00)
--> #2A((2 8 3) (H 1 4) (7 6 5)) (H=3.00)
--> #2A((2 H 3) (1 8 4) (7 6 5)) (H=3.00)
--> #2A((2 8 3) (1 4 H) (7 6 5)) (H=4.00)
--> #2A((2 8 3) (1 6 4) (7 H 5)) (H=4.00)
NIL
```

8–puzzle por escalada con la primera heurística



Segunda heurística en el problema del 8–puzzle




```
(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 con la segunda heurística

```
> (load "p-8-puzzle")
T
> (load "b-escalada")
T
> (defun heuristica (estado) (heuristica-2 estado))
HEURISTICA
> (busqueda-en-escalada-con-traza :traza 2)
1: #2A((2 8 3) (1 6 4) (7 H 5)) (H=5.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=6.00)
--> #2A((2 8 3) (1 H 4) (7 6 5)) (H=4.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=6.00)
2: #2A((2 8 3) (1 H 4) (7 6 5)) (H=4.00)
--> #2A((2 8 3) (H 1 4) (7 6 5)) (H=5.00)
--> #2A((2 H 3) (1 8 4) (7 6 5)) (H=3.00)
--> #2A((2 8 3) (1 4 H) (7 6 5)) (H=5.00)
--> #2A((2 8 3) (1 6 4) (7 H 5)) (H=5.00)
```

8–puzzle por escalada con la segunda heurística

```
3: #2A((2 H 3) (1 8 4) (7 6 5)) (H=3.00)
--> #2A((H 2 3) (1 8 4) (7 6 5)) (H=2.00)
--> #2A((2 3 H) (1 8 4) (7 6 5)) (H=4.00)
--> #2A((2 8 3) (1 H 4) (7 6 5)) (H=4.00)
4: #2A((H 2 3) (1 8 4) (7 6 5)) (H=2.00)
--> #2A((2 H 3) (1 8 4) (7 6 5)) (H=3.00)
--> #2A((1 2 3) (H 8 4) (7 6 5)) (H=1.00)
5: #2A((1 2 3) (H 8 4) (7 6 5)) (H=1.00)
--> #2A((H 2 3) (1 8 4) (7 6 5)) (H=2.00)
--> #2A((1 2 3) (8 H 4) (7 6 5)) (H=0.00)
--> #2A((1 2 3) (7 8 4) (H 6 5)) (H=2.00)
6: #2A((1 2 3) (8 H 4) (7 6 5)) (H=0.00)
Estado inicial:          #2A((2 8 3) (1 6 4) (7 H 5))
Estado final:            #2A((1 2 3) (8 H 4) (7 6 5))
Solucion:                (MOVER-ARRIBA MOVER-ARRIBA MOVER-IZQUIERDA
                          MOVER-ABAJO MOVER-DERECHA)
```

Comparación de escalada en el 8–puzzle

Estado inicial	Profundidad iterativa: Tiempo (seg.)	Profundidad iterativa: Espacio (bytes)	Escalada: Tiempo (seg.)	Escalada: Espacio (bytes)
283 164 7 5	0.06	9.028	0.05	6.516
481 3 2 765	15.88	656.020	0.11	14.604

Propiedades de la búsqueda por escalada

- Complejidad:
  - r: factor de ramificación.
  - p: profundidad de la solución.
  - Complejidad en espacio: $O(1)$.
  - Complejidad en tiempo: $O(r^p)$.
- 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))
```

El 8–puzzle por primero el mejor

```
> (busqueda-por-primero-el-mejor-con-traza :traza 2)
1: #2A((2 8 3) (1 6 4) (7 H 5)) (H=4.00)
--> #2A((2 8 3) (1 H 4) (7 6 5)) (H=3.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=5.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=5.00)
2: #2A((2 8 3) (1 H 4) (7 6 5)) (H=3.00)
--> #2A((2 8 3) (1 4 H) (7 6 5)) (H=4.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=5.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=5.00)
3: #2A((2 8 3) (H 1 4) (7 6 5)) (H=3.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=5.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=5.00)
```

El 8–puzzle por primero el mejor

```
4: #2A((2 H 3) (1 8 4) (7 6 5)) (H=3.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=5.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=5.00)
5: #2A((H 2 3) (1 8 4) (7 6 5)) (H=2.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=5.00)
6: #2A((1 2 3) (H 8 4) (7 6 5)) (H=1.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=5.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=5.00)
Estado inicial:          #2A((2 8 3) (1 6 4) (7 H 5))
Estado final:            #2A((1 2 3) (8 H 4) (7 6 5))
Solucion:                (MOVER-ARRIBA MOVER-ARRIBA MOVER-IZQUIERDA
                          MOVER-ABAJO MOVER-DERECHA)
```

El 8–puzzle por primero el mejor

```
> (defun heuristica (estado) (heuristica-2 estado))
HEURISTICA
> (busqueda-por-primero-el-mejor-con-traza :traza 2)
1: #2A((2 8 3) (1 6 4) (7 H 5)) (H=5.00)
--> #2A((2 8 3) (1 H 4) (7 6 5)) (H=4.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=6.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=6.00)
2: #2A((2 8 3) (1 H 4) (7 6 5)) (H=4.00)
--> #2A((2 8 3) (1 4 H) (7 6 5)) (H=5.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=6.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=6.00)
3: #2A((2 H 3) (1 8 4) (7 6 5)) (H=3.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=6.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=6.00)
```

El 8–puzzle por primero el mejor

```
4: #2A((H 2 3) (1 8 4) (7 6 5)) (H=2.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=6.00)
5: #2A((1 2 3) (H 8 4) (7 6 5)) (H=1.00)
--> #2A((2 8 3) (1 6 4) (H 7 5)) (H=6.00)
--> #2A((2 8 3) (1 6 4) (7 5 H)) (H=6.00)
Estado inicial:          #2A((2 8 3) (1 6 4) (7 H 5))
Estado final:            #2A((1 2 3) (8 H 4) (7 6 5))
Solucion:                (MOVER-ARRIBA MOVER-ARRIBA MOVER-IZQUIERDA
                          MOVER-ABAJO MOVER-DERECHA)
```

Comparación de primero el mejor en el 8–puzzle

Estado inicial	Profundidad iterativa: Tiempo (seg.)	Profundidad iterativa: Espacio (bytes)	Escalada: Tiempo (seg.)	Escalada: Espacio (bytes)	Primero el mejor: Tiempo (seg.)	Primero el mejor: Espacio (bytes)
283 164 7 5	0.06	9.028	0.05	6.516	0.05	6.884
481 3 2 765	15.88	656.020	0.11	14.604	0.13	15.844

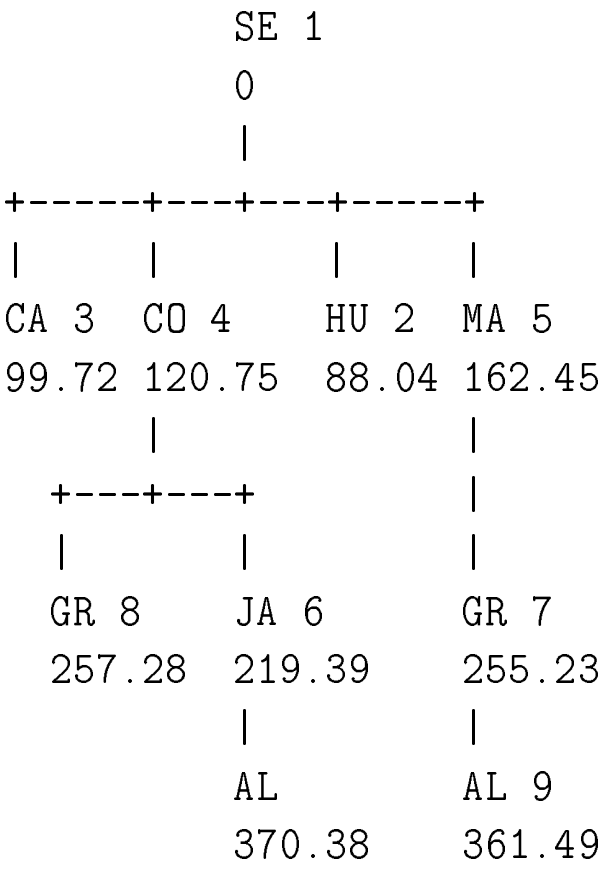
Propiedades de la búsqueda por primero el mejor

- 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
```

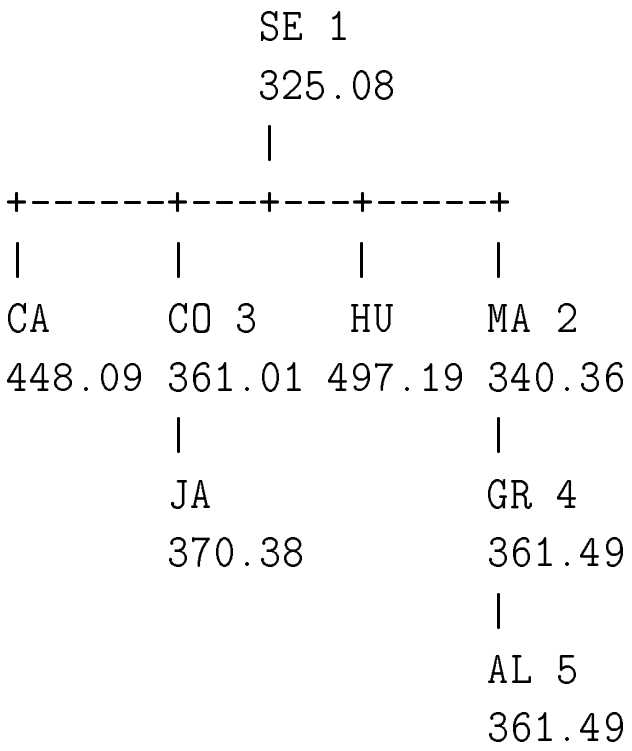
El viaje mediante búsqueda optimal

```
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* (I) 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* (I)

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 y cuyo coste y coste más heurística 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 A* (I)

```
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* (I)

```
(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* (I)

```
(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* (I)

```
(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))))))
```

Implementación de A* (I)

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

El viaje mediante A* (I)

```
> (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
```

El viaje mediante A* (I)

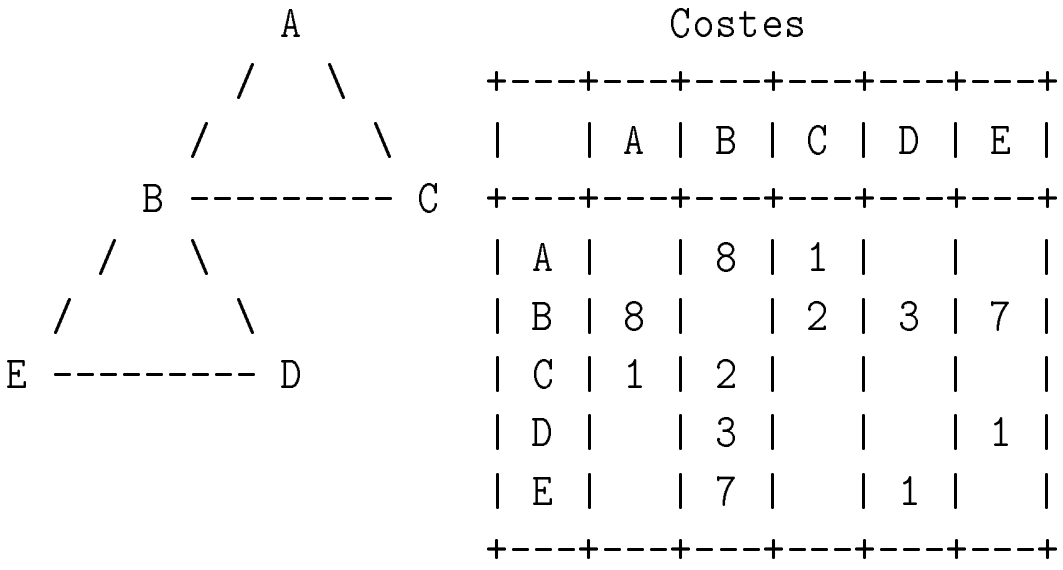
```
#S(NODO-HC
   :ESTADO ALMERIA
   :CAMINO (IR-A-ALMERIA IR-A-GRANADA IR-A-MALAGA)
   :COSTE-CAMINO 361.48953
   :COSTE-MAS-HEURISTICA 361.48953))
```

Propiedades de A* (I)

- 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))
```

Representación del problema del grafo

```
(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)))
```

Representación del problema del grafo

```
(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)))))
```

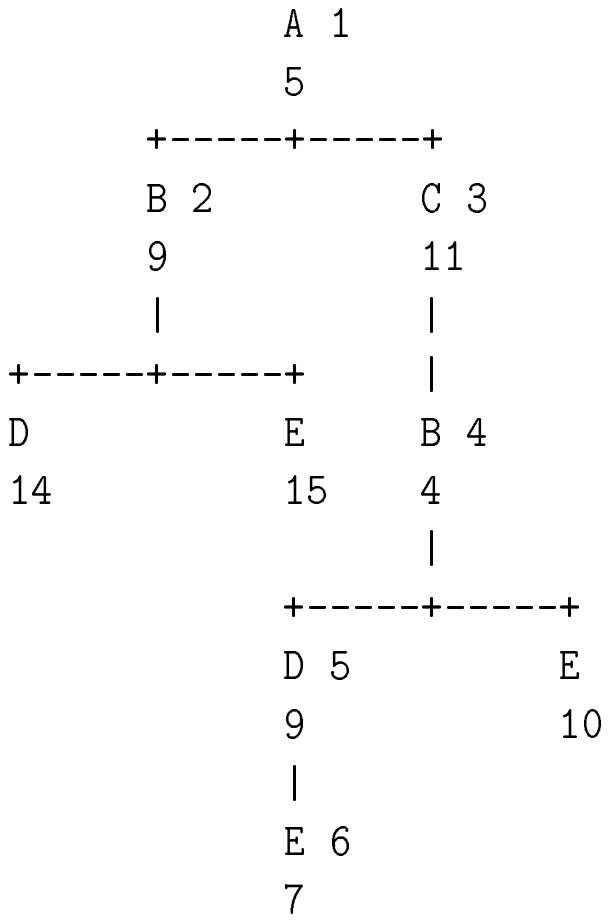
Solución del grafo mediante A* (I)

```
> (load "p-grafo-a-estrella.lsp")
T
> (load "b-a-estrella-1.lsp")
T
> (busqueda-a-estrella-1-con-traza)
A Coste+Heuristica: 5.00
  B Coste+Heuristica: 9.00
  C Coste+Heuristica: 11.00
B Coste+Heuristica: 9.00
  D Coste+Heuristica: 14.00
  E Coste+Heuristica: 15.00
C Coste+Heuristica: 11.00
  B Coste+Heuristica: 4.00
```

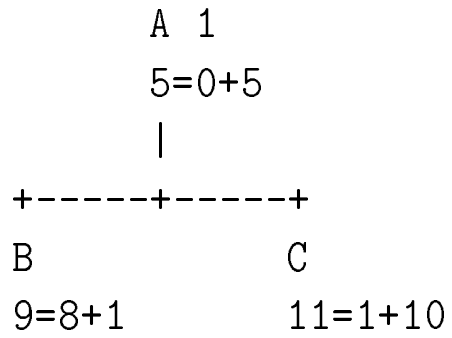
Solución del grafo mediante A* (I)

```
B Coste+Heuristica: 4.00
  D Coste+Heuristica: 9.00
  E Coste+Heuristica: 10.00
D Coste+Heuristica: 9.00
  E Coste+Heuristica: 7.00
#S(NODO-HC
   :ESTADO E
   :CAMINO (IR-A-E IR-A-D IR-A-B IR-A-C)
   :COSTE-CAMINO 7
   :COSTE-MAS-HEURISTICA 7)
```

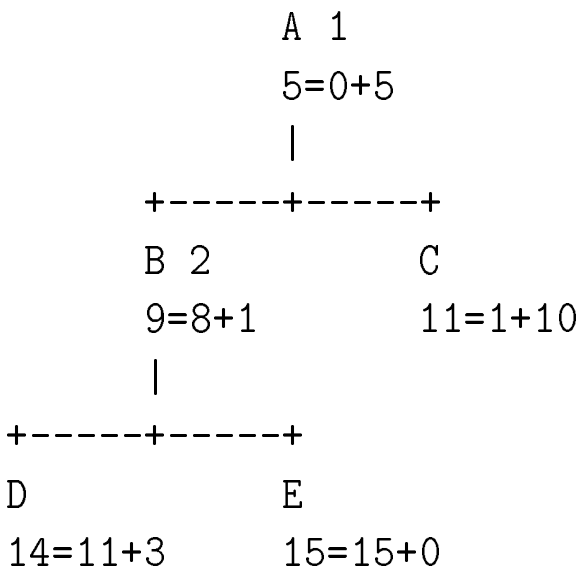
Solución del grafo mediante A* (I)



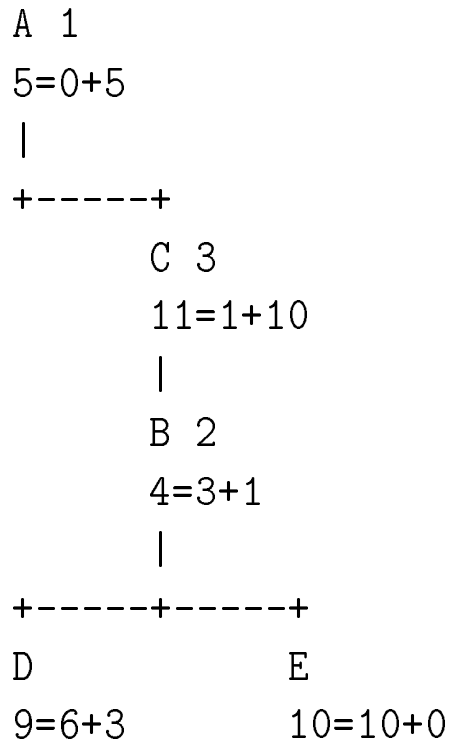
Solución del grafo mediante A*



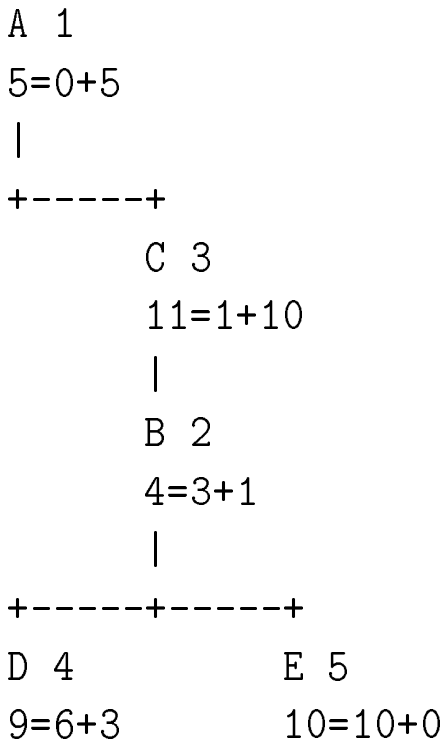
Solución del grafo mediante A*



Solución del grafo mediante A*



Solución del grafo mediante A*



Camino de la solución: A C B E

Solución del grafo mediante A*

```
> (load "p-grafo-a-estrella")
T
> (load "b-a-estrella")
T
> (busqueda-a-estrella-con-traza)
Actual <A 5.00=0.00+5.00 0.00>
  Nuevo <B 9.00=8.00+1.00 8.00>
  Nuevo <C 11.00=1.00+10.00 1.00>
Actual <B 9.00=8.00+1.00 8.00>
  Viejo abierto <C 11.00=1.00+10.00 1.00>
  Nuevo <D 14.00=11.00+3.00 3.00>
  Nuevo <E 15.00=15.00+0.00 7.00>
```

Solución del grafo mediante A*

```
Actual <C 11.00=1.00+10.00 1.00>
  Viejo cerrado <B 4.00=3.00+1.00 2.00>
    Mejora   B-<E 15.00=15.00+0.00 7.00>
    Mejorado B-<E 10.00=10.00+0.00 7.00>
    Mejora   B-<D 14.00=11.00+3.00 3.00>
    Mejorado B-<D 9.00=6.00+3.00 3.00>
Actual <D 9.00=6.00+3.00 3.00>
  Viejo abierto <E 7.00=7.00+0.00 1.00>
<E 7.00=7.00+0.00 1.00>  ;
(IR-A-C IR-A-B IR-A-D IR-A-E)
```

Definición de nodo A*

- Los nodos para el procedimiento A* se componen de:
  - un estado
  - el padre del nodo
  - el último operador
  - el coste del aplicar el último operador al padre del nodo
  - el coste del camino actual
  - el valor de la función de evaluación heurística del estado actual el coste total estimado (la suma del coste hasta el nodo y la estimación del coste restante)
  - los sucesores del nodo actual

Definición de nodo A*

```
(defstruct (nodo-a (:constructor crea-nodo-a)
   (:conc-name nil)
   (:print-function escribe-nodo-a))
  estado
  padre
  operador
  coste-transicion
  coste-camino
  heuristica-del-nodo
  coste-total-estimado
  sucesores)

(defun escribe-nodo-a (nodo &optional (canal t) profundidad)
  (format canal "<~a ~,2f=~,2f+~,2f ~,2f> "
  (estado nodo) (coste-total-estimado nodo) (coste-camino nodo)
  (heuristica-del-nodo nodo) (coste-transicion nodo)))
```

Procedimiento A*

```
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;
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. VIEJO-ABIERTO y
1.5. VIEJO-CERRADO.
2. Mientras que ABIERTOS no está vacía,
2.1. tomar como ACTUAL el primero de abiertos (que el nodo expandido no
     analizado de menor coste estimado),
2.2. quitarlo de ABIERTOS y
2.3. colocarlo en CERRADOS.
2.4. Si ACTUAL es un nodo final,
2.4.1. devolver el nodo ACTUAL y la correspondiente solución;
```

Procedimiento A*

```
      Si ACTUAL es un nodo final,
2.4.2. para cada uno de los sucesores del nodo ACTUAL hacer lo siguiente:
2.4.2.1. si hay en ABIERTOS un nodo con el mismo estado que el sucesor,
         llamar a dicho nodo VIEJO-ABIERTO y, si el sucesor es mejor que
         VIEJO-ABIERTO, ponerlo en su lugar;
2.4.2.2. si hay en CERRADOS un nodo con el mismo estado que el sucesor,
         llamar a dicho nodo VIEJO-CERRADO y, si el sucesor es mejor que
         VIEJO-CERRADO, ponerlo en su lugar y actualizar los descendientes
         de VIEJO-CERRADO;
2.4.2.3. en caso contrario, añadir el sucesor a la lista de los sucesores
         del nodo ACTUAL y a la lista de ABIERTOS.
2.4.3. ordenar ABIERTOS por el coste total estimado.
3. Si ABIERTOS está vacía, devolver NIL.
```

Procedimiento A*

```
(defun busqueda-a-estrella ()
  (let ((abiertos (list (nodo-inicial)))                    ;1.1
        (cerrados nil)                                      ;1.2
        (actual nil)                                        ;1.3
        (viejo-abierto nil)                                 ;1.4
        (viejo-cerrado nil))                                ;1.5
    (loop until (null abiertos) do                          ;2
      (setf actual (first abiertos)                         ;2.1
            abiertos (rest abiertos)                        ;2.2
            cerrados (cons actual cerrados))                ;2.3
```

Procedimiento A*

```
(cond ((es-estado-final (estado actual))                          ;2.4
       (return (values actual                                     ;2.4.1
                       (nreverse (forma-camino actual)))))
      (t (loop for sucesor in (lista-sucesores actual) do         ;2.4.2
           (cond ((setf viejo-abierto (esta sucesor abiertos))    ;2.4.2.1
                  (when (es-mejor sucesor viejo-abierto)
                        (cambia viejo-abierto sucesor)))
                 ((setf viejo-cerrado (esta sucesor cerrados))    ;2.4.2.2
                  (when (es-mejor sucesor viejo-cerrado)
                        (cambia viejo-cerrado sucesor)
                        (actualiza-descendientes viejo-cerrado)))
                 (t (setf (sucesores actual)                      ;2.4.2.3
                          (cons sucesor (sucesores actual)))
                    (setf abiertos (cons sucesor abiertos)))))
         (setf abiertos                                           ;2.4.3
               (ordena-por-coste-total-estimado abiertos))))))
```

Procedimiento A*

```
(defun nodo-inicial ()
  (crea-nodo-a :estado               *estado-inicial*
               :padre                nil
               :coste-transicion     0
               :coste-camino         0
               :heuristica-del-nodo  (heuristica *estado-inicial*)
               :coste-total-estimado (heuristica *estado-inicial*)
               :sucesores            nil))
```

Procedimiento A*

```
(defun forma-camino (nodo &optional (res nil))
  (if (null (padre nodo))
      (nreverse res)
      (forma-camino (padre nodo) (cons (operador nodo) res))))

(defun lista-sucesores (nodo)
  (let ((resultado ()))
    (loop for operador in *operadores* do
      (let ((siguiente (sucesor nodo operador)))
        (when siguiente (push siguiente resultado))))
    (nreverse resultado)))
```

Procedimiento A*

```
(defun sucesor (nodo operador)
  (let ((siguiente-estado (aplica operador (estado nodo))))
    (when siguiente-estado
          (let* ((coste-transicion (coste-de-aplicar-operador
                                    (estado nodo) operador))
                 (heuristica-del-nodo (heuristica siguiente-estado))
                 (coste-camino (+ coste-transicion (coste-camino nodo)))
                 (coste-total-estimado (+ coste-camino heuristica-del-nodo)))
            (crea-nodo-a
             :estado              siguiente-estado
             :padre               nodo
             :operador            operador
             :coste-transicion    coste-transicion
             :heuristica-del-nodo heuristica-del-nodo
             :coste-camino        coste-camino
             :coste-total-estimado coste-total-estimado)))))
```

Procedimiento A*

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

(defun es-mejor (sucesor viejo)
  (< (coste-camino sucesor)
     (coste-camino viejo)))
```

Procedimiento A*

```
(defun cambia (viejo sucesor)
  (setf (padre viejo)                (padre sucesor)
        (operador viejo)             (operador sucesor)
        (coste-transicion viejo)     (coste-transicion sucesor)
        (coste-camino viejo)         (coste-camino sucesor)
        (coste-total-estimado viejo) (coste-total-estimado sucesor))
  (setf (sucesores (padre sucesor))  (cons viejo (sucesores (padre sucesor))))
  (delete viejo (sucesores (padre viejo)) :test #'equalp :count 1))
```

Lisp: Borrado

```
* (DELETE ELEMENTO LISTA &KEY :TEST :COUNT)
    (delete 4 '(1 2 4 1 3 4 5))                      => (1 2 1 3 5)
    (delete 3 '(1 2 4 1 3 4 5) :test #'>)            => (4 3 4 5)
    (delete 4 '(1 2 4 1 3 4 5) :count 1)             => (1 2 1 3 4 5)
    (delete 3 '(1 2 4 1 3 4 5) :test #'> :count 2) => (4 1 3 4 5)
```

Procedimiento A*

```
(defun actualiza-descendientes (viejo)
  (loop for nodo in (sucesores viejo) do
    (when (< (+ (coste-camino viejo) (coste-transicion nodo))
             (coste-camino nodo))
          (setf (coste-camino nodo)
                (+ (coste-camino viejo) (coste-transicion nodo)))
          (setf (coste-total-estimado nodo)
                (+ (heuristica-del-nodo nodo) (coste-camino nodo)))
          (actualiza-descendientes nodo))))

(defun ordena-por-coste-total-estimado (lista)
  (sort lista #'< :key #'coste-total-estimado))
```

El viaje mediante A*

```
> (load "p-viaje")
T
> (load "b-a-estrella.lsp")
T
> (busqueda-a-estrella-con-traza)
Actual <SEVILLA 325.08=0.00+325.08 0.00>
  Nuevo <CADIZ 448.09=99.72+348.37 99.72>
  Nuevo <CORDOBA 361.01=120.75+240.27 120.75>
  Nuevo <HUELVA 497.19=88.04+409.15 88.04>
  Nuevo <MALAGA 340.36=162.45+177.91 162.45>
Actual <MALAGA 340.36=162.45+177.91 162.45>
  Viejo abierto <CADIZ 448.09=99.72+348.37 99.72>
  Viejo abierto <CORDOBA 361.01=120.75+240.27 120.75>
  Nuevo <GRANADA 361.49=255.23+106.26 92.78>
```

El viaje mediante A*

```
Actual <CORDOBA 361.01=120.75+240.27 120.75>
  Viejo abierto <GRANADA 361.49=255.23+106.26 92.78>
  Nuevo <JAEN 370.38=219.39+150.99 98.65>
Actual <GRANADA 361.49=255.23+106.26 92.78>
  Nuevo <ALMERIA 361.49=361.49+0.00 106.26>
  Viejo abierto <JAEN 370.38=219.39+150.99 98.65>
<ALMERIA 361.49=361.49+0.00 106.26>  ;
(IR-A-MALAGA IR-A-GRANADA IR-A-ALMERIA)
```

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 completo.
- Es optimal, si la heurística es optimista

Problema de misioneros y caníbales

- Enunciado:
  - Tenemos M misioneros y C caníbales en la orilla izquierda de un río, y queremos pasarlos a la otra orilla ($M \geq C$).
  - Disponemos de una barca en la que caben a lo sumo B personas ($B \leq C + M$).
  - En ningún momento el número de caníbales que hay en la orilla en que no está la barca puede superar al número de misioneros e igualmente en la barca.
  - Se pide hallar un plan de viajes de la barca entre una orilla y otra de manera que se consiga trasladar a todas las personas a la orilla de destino y respetando las restricciones anteriores.

Planteamiento del problema de misioneros y caníbales

- Representación de estados: (X Y Z), donde X es el número de misioneros que hay en la orilla izquierda, Y el de caníbales y Z la posición de la barca (1=izquierda, 0=derecha).
- Número de estados: $M \times C \times 2$.
- Estado inicial: (M C 1)
- Estado final: (0 0 0)
- Operadores:
  Mover N1 misioneros y N2 caníbales a la orilla opuesta.

Solución de misioneros en profundidad

```
> (load "p-misioneros")
T
> (load "b-profundidad")
T
> (misioneros 3 3 2)
(3 3 2)
> *operadores*
((MOVER 2 0) (MOVER 1 1) (MOVER 0 2) (MOVER 1 0) (MOVER 0 1))
```

Solución de misioneros en profundidad

```
> (escribe-solucion (nreverse (camino (busqueda-en-profundidad))))
3 3 B || 0 0
2 2   || 1 1 B
3 2 B || 0 1
2 1   || 1 2 B
3 1 B || 0 2
1 1   || 2 2 B
1 3 B || 2 0
0 2   || 3 1 B
0 3 B || 3 0
0 1   || 3 2 B
1 1 B || 2 2
0 0   || 3 3 B
NIL
```

Solución de misioneros en profundidad

```
> (misioneros 4 4 3)
(4 4 3)
> (escribe-solucion (nreverse (camino (busqueda-en-profundidad))))
4 4 B || 0 0
3 2   || 1 2 B
4 3 B || 0 1
2 2   || 2 2 B
2 4 B || 2 0
0 3   || 4 1 B
1 4 B || 3 0
0 2   || 4 2 B
2 2 B || 2 2
0 1   || 4 3 B
1 1 B || 3 3
0 0   || 4 4 B
NIL
```

Implementación del problema de los misioneros

- Problema genérico:

```
(defvar *misioneros*)
(defvar *canibales*)
(defvar *capacidad*)

(defun misioneros (misioneros canibales capacidad)
  (setf *misioneros* misioneros
        *canibales* canibales
        *capacidad* capacidad)
  (crea-estado-inicial)
  (crea-operadores)
  (list misioneros canibales capacidad))
```

Implementación del problema de los misioneros

- Representación de estados:

```
(defun orilla-de-la-barca (estado)
  (if (= (third estado) 1)
      'izquierda
      'derecha))

(defun orilla-opuesta (estado)
  (if (= (third estado) 1)
      0
      1))
```

Implementación del problema de los misioneros

```
(defun misioneros-en-izquierda (estado)
  (first estado))

(defun canibales-en-izquierda (estado)
  (second estado))

(defun misioneros-en-derecha (estado)
  (- *misioneros* (misioneros-en-izquierda estado)))

(defun canibales-en-derecha (estado)
  (- *canibales* (canibales-en-izquierda estado)))
```

Implementación del problema de los misioneros

- Estado inicial:

```
(defun crea-estado-inicial ()
  (defparameter *estado-inicial*
    (list *misioneros* *canibales* 1)))
```

- Estado final:

```
(defparameter *estado-final*
  (list 0 0 0))

(defun es-estado-final (estado)
  (equal estado *estado-final*))
```

Implementación del problema de los misioneros

- Operadores:

```
(defun crea-operadores ()
  (defparameter *operadores* nil)
    (loop for m from 0 to *misioneros* do
      (loop for c from 0 to *canibales* do
        (when (and (<= 1 (+ m c) *capacidad*)
                   ;(>= m c))
                   )
              (push (list 'mover m c) *operadores*))))
    (ordena *operadores*))
```

Implementación del problema de los misioneros

```
(defun ordena (operadores)
  (sort operadores #'> :key #'movidos))

(defun movidos (operador)
  (+ (second operador)
     (third operador)))
```

Implementación del problema de los misioneros

```
(defun mover (misioneros canibales estado)
  (let* ((operacion
          (if (eq (orilla-de-la-barca estado) 'izquierda) '- '+))
         (misioneros-en-izquierda (misioneros-en-izquierda estado))
         (canibales-en-izquierda (canibales-en-izquierda estado))
         (orilla-opuesta (orilla-opuesta estado))
         (siguiente
          (list (funcall operacion misioneros-en-izquierda
                         misioneros)
                (funcall operacion canibales-en-izquierda
                         canibales)
                orilla-opuesta)))
    (when (es-posible siguiente) siguiente)))
```

Implementación del problema de los misioneros

```
(defun es-posible (estado)
  (and (<= 0 (misioneros-en-izquierda estado) *misioneros*)
       (<= 0 (canibales-en-izquierda estado) *canibales*)
       (if (eq (orilla-de-la-barca estado) 'derecha)
           (not (< 0
                   (misioneros-en-izquierda estado)
                   (canibales-en-izquierda estado)))
           (not (< 0
                   (misioneros-en-derecha estado)
                   (canibales-en-derecha estado))))))

(defun aplica (operador estado)
  (funcall (symbol-function (first operador))
           (second operador)
           (third operador)
           estado))
```

Implementación del problema de los misioneros

- Auxiliares

```
(defun escribe-solucion (camino)
  (let ((estado *estado-inicial*))
    (escribe-estado estado)
    (loop for operador in camino do
      (setf estado (aplica operador estado))
      (escribe-estado estado))))

(defun escribe-estado (estado)
  (format t "~&~a ~a ~a || ~a ~a ~a"
          (misioneros-en-izquierda estado)
          (canibales-en-izquierda estado)
          (if (eq (orilla-de-la-barca estado) 'izquierda) "B" " ")
          (misioneros-en-derecha estado)
          (canibales-en-derecha estado)
          (if (eq (orilla-de-la-barca estado) 'derecha) "B" " ")))
```

Implementación del problema de los misioneros

```
(defun coste-de-aplicar-operador (estado operador)
  (+ (second operador)
     (third operador)))

(defun heuristica-1 (estado)
  (+ (misioneros-en-izquierda estado)
     (canibales-en-izquierda estado)))

(defun heuristica-2 (estado)
  (* (+ (misioneros-en-izquierda estado)
        (canibales-en-izquierda estado))
     *capacidad*))
```

Comparación de soluciones de los misioneros

misioneros con 3 misioneros 3 canibales y capacidad 2			
Procedimiento	Coste	Segundos	Bytes
en-anchura	14	0.02	5,748
en-profundidad	18	0.01	3,148
en-profundidad-iterativa	14	0.05	19,212
optimal	14	0.02	6,200
H=1 en-escalada	Sin sol		
H=1 por-primero-el-mejor	14	0.02	4,728
H=1 a-estrella-1	14	0.02	6,492
H=1 a-estrella	14	0.02	6,716
H=2 en-escalada	Sin sol		
H=2 por-primero-el-mejor	14	0.02	4,764
H=2 a-estrella-1	14	0.02	6,400
H=2 a-estrella	14	0.02	6,716

Comparación de soluciones de los misioneros

misioneros con 4 misioneros 4 canibales y capacidad 3			
Procedimiento	Coste	Segundos	Bytes
en-anchura	16	0.04	14,936
en-profundidad	26	0.02	5,192
en-profundidad-iterativa	22	0.13	41,488
optimal	14	0.10	16,252
H=1 en-escalada	Sin sol		
H=1 por-primero-el-mejor	14	0.04	10,004
H=1 a-estrella-1	14	0.08	21,664
H=1 a-estrella	14	0.05	15,672
H=2 en-escalada	Sin sol		
H=2 por-primero-el-mejor	14	0.04	10,004
H=2 a-estrella-1	14	0.04	11,440
H=2 a-estrella	14	0.04	8,356

Comparación de soluciones de los misioneros

misioneros con 10 misioneros 10 canibales 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-1	26	2.20	292,316
H=1 a-estrella	26	0.68	124,704
H=2 en-escalada	Sin sol		
H=2 por-primero-el-mejor	26	0.21	42,044
H=2 a-estrella-1	26	0.28	48,432
H=2 a-estrella	26	0.21	37,744

Comparación de soluciones de los misioneros

misioneros con 20 misioneros 20 canibales 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-1	46	43.48	3,090,536
H=1 a-estrella	46	8.25	916,840
H=2 en-escalada	Sin sol		
H=2 por-primero-el-mejor	46	1.42	188,804
H=2 a-estrella-1	46	1.90	220,916
H=2 a-estrella	46	1.33	156,076

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".
- [Mira-95]
  Cap. 4: "Búsqueda heurística".
- [Rich-91]
  Cap. 3: "Técnicas de búsqueda heurística".
- [Winston-94]
  Cap. 4: "Redes y búsqueda básica" y
  Cap. 5: "Redes y búsqueda óptima".