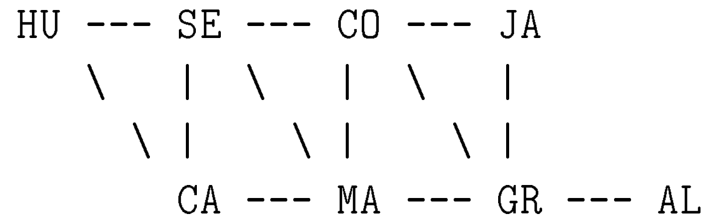
Tema 4:
Resolución de problemas mediante búsqueda heurística

Búsqueda heurística

- Idea fundamental de la búsqueda heurística:
  - Podar la búsqueda
  - Comparación de los estados
- Función de evaluación heurística:
  - Estima la distancia al final
  - Valor en el estado final: 0
  - Comparación de los estados

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))
```

- Coste de un camino: suma de distancias.

- Función de evaluación heurística:

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

Lisp

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

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

* (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

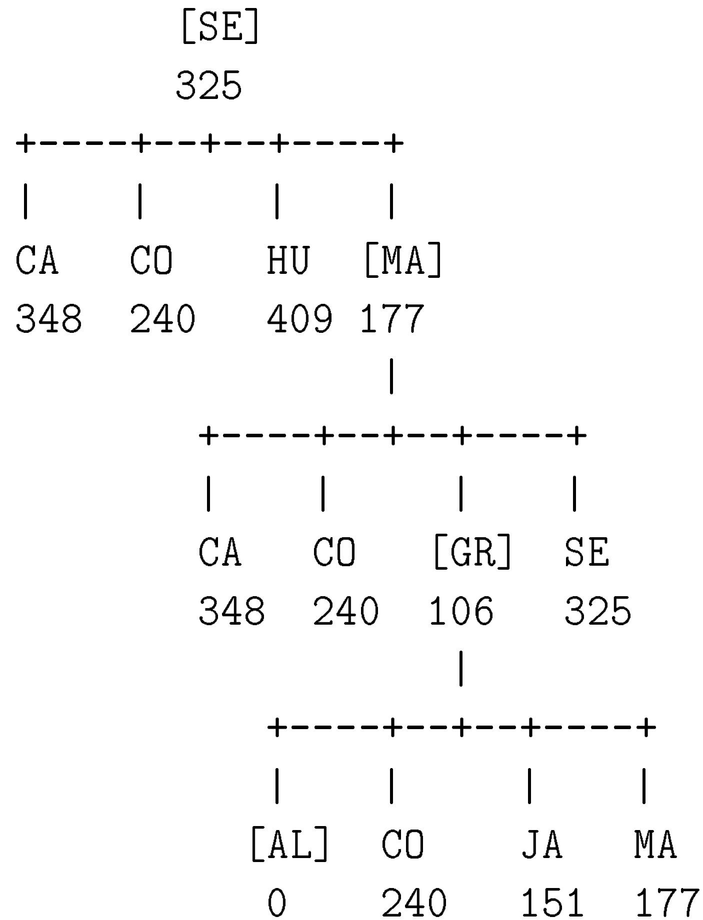
- Coordenadas:

```
(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))))
```

Implementación del problema de viaje con heurística

```
(defun coste-de-aplicar-operador (estado operador)
  (let ((estado-sucesor (funcall (symbol-function operador) estado)))
    (when estado-sucesor (distancia estado estado-sucesor))))

(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*))))

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

Grafo de escalada para el problema del viaje



Procedimiento de escalada

1. Si el estado actual es un estado final, entonces parar y devolverlo.
2. En caso contrario, aplicar todos los operadores al estado actual.
   2.1. Si alguno de los nuevos estados es mejor que el actual, entonces tomar como actual el mejor de ellos.
   2.2. En caso contrario, terminar con fallo.

Implementación de escalada

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

(defun escalada ()
  (let ((actual (crea-nodo-h :estado *estado-inicial*
                             :camino nil
                             :coste-camino 0
                             :heuristica-del-nodo (heuristica *estado-inicial*))))
    (loop until (null actual) do
      (cond ((es-estado-final (estado actual))
             (return actual))
            (t (setf actual (mejor (sucesores actual) (heuristica-del-nodo actual))))))))
```

Implementación de escalada

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

(defun sucesor (nodo operador)
  (let ((siguiente-estado (funcall (symbol-function operador)
                                   (estado nodo))))
    (when siguiente-estado
          (crea-nodo-h :estado siguiente-estado
                       :camino (cons operador (camino nodo))
                       :heuristica-del-nodo (heuristica siguiente-estado)
                       :coste-camino
                       (+ (coste-de-aplicar-operador (estado nodo) operador)
                          (coste-camino nodo))))))
```

Implementación de 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))
```

Comparación de anchura y escalada para el viaje

```
                                   Anchura  Escalada
- Tiempo:
  - Numero de nodos visitados:           8         4
  - Tiempo de ejecucion (seg):         0.2      0.09
- Espacio:
  - Maximo en abiertos:                  4         1
  - Bytes:                            4700      2764
```

Primera heurística en el problema del 8-puzzle

- Coste:

```
(defun coste-de-aplicar-operador (estado operador)
  1)
```

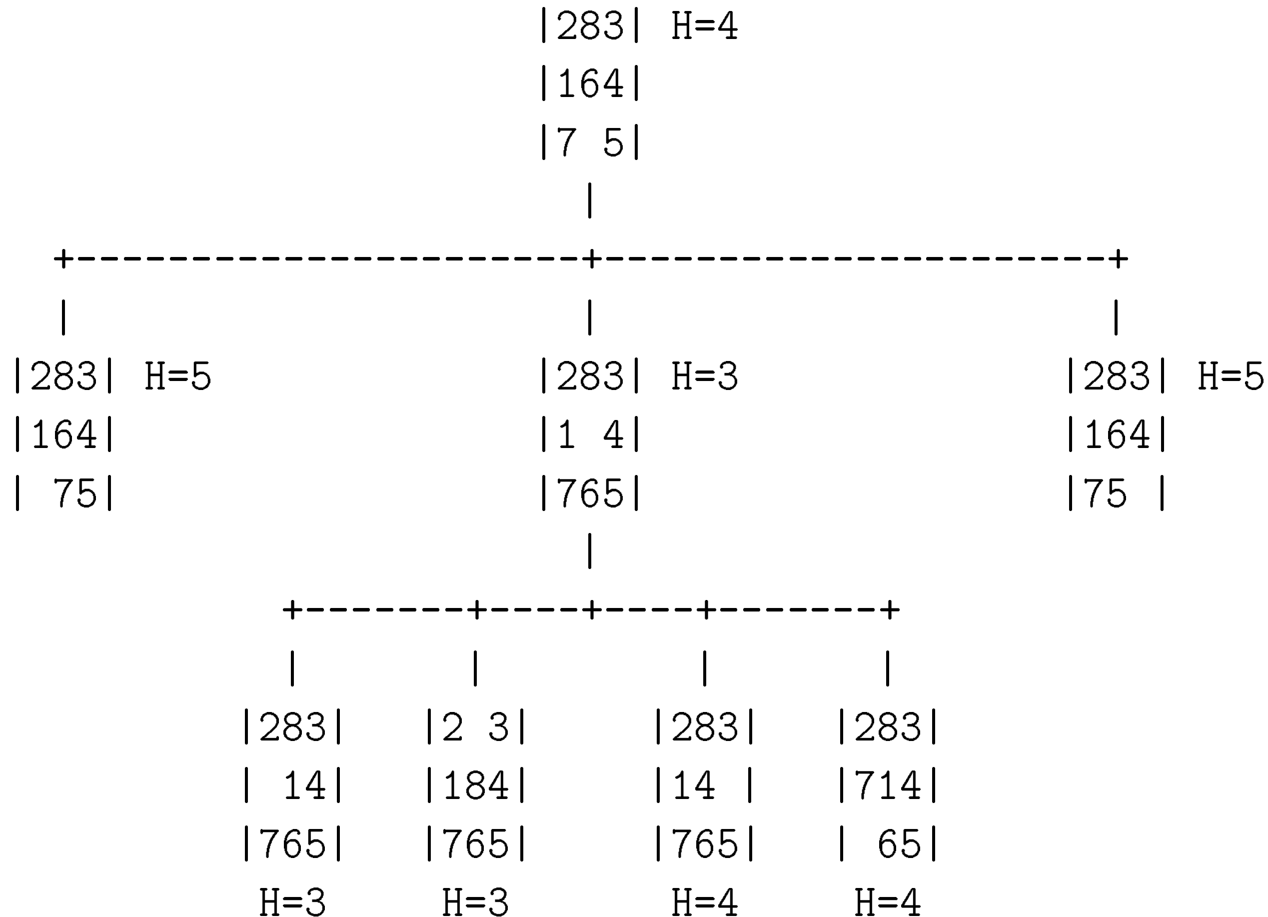
- Heurística:

```
(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 "8-puzzle")
T
> (load "busq-heu")
T
> (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 (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 "8-puzzle")
T
> (load "busq-heu")
T
> (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	1.68	25.252	0.97	7.704
481 3 2 765	424.81	5.802.744	2.12	17.324

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 optimal
- 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)
```

Búsqueda por primero el mejor

```
(defun primero-el-mejor ()
  (let ((abiertos (list (crea-nodo-h :estado *estado-inicial*
                                     :camino nil
                                     :coste 0
                                     :heuristica-del-nodo
                                     (heuristica *estado-inicial*))))
        (cerrados nil)
        (actual nil)
        (nuevos-sucesores nil))
```

Búsqueda por primero el mejor

```
(loop until (null abiertos) do
      (setf actual (first abiertos))
      (setf abiertos (rest abiertos))
      (setf cerrados (cons actual cerrados))
      (cond ((es-estado-final (estado actual))
             (return actual))
            (t (setf nuevos-sucesores
                     (nuevos-sucesores actual abiertos cerrados))
               (setf abiertos
                     (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 con primera heurística

```
> (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)
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)
```

El 8-puzzle por primero el mejor con primera heurística

```
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 con segunda heurística

```
> (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)
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)
```

El 8-puzzle por primero el mejor con segunda heurística

```
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	1.68	25.252	0.9	7.704	1.92	16.188
481 3 2 765	424.81	5.802.744	2.12	17.324	4.43	39.580

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 optimal

Procedimiento de búsqueda optimal

```
(defun busqueda-optimal
  (let ((abiertos (list (crea-nodo-h :estado *estado-inicial*
                                     :camino nil
                                     :coste 0
                                     :heuristica-del-nodo
                                     (heuristica *estado-inicial*))))
        (cerrados nil)
        (actual nil)
        (nuevos-sucesores nil))
```

Procedimiento de búsqueda optimal

```
    (loop until (null abiertos) do
          (setf actual (first abiertos))
          (setf abiertos (rest abiertos))
          (setf cerrados (cons actual cerrados))
          (cond ((es-estado-final (estado actual))
                 (return actual))
                (t (setf nuevos-sucesores
                         (nuevos-sucesores actual abiertos cerrados))
                   (setf abiertos
                         (ordena-por-coste
                           (append abiertos nuevos-sucesores)))))))

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

El viaje mediante búsqueda optimal

```
> (load "viaje")
T
> (load "busq-heu")
T
> (busqueda-optimal-con-traza :traza 2)
1: SEVILLA (H=325.08 C=0.00)
--> HUELVA (H=409.15 C=88.04)
--> CADIZ (H=348.37 C=99.72)
--> CORDOBA (H=240.27 C=120.75)
--> MALAGA (H=177.91 C=162.45)
2: HUELVA (H=409.15 C=88.04)
3: CADIZ (H=348.37 C=99.72)
4: CORDOBA (H=240.27 C=120.75)
--> JAEN (H=150.99 C=219.39)
--> GRANADA (H=106.26 C=257.28)
```

El viaje mediante búsqueda optimal

```
5: MALAGA (H=177.91 C=162.45)
6: JAEN (H=150.99 C=219.39)
--> ALMERIA (H=0.00 C=370.38)
7: GRANADA (H=106.26 C=257.28)
Estado inicial:          SEVILLA
Estado final:            ALMERIA
Solucion:                (IR-A-CORDOBA IR-A-JAEN IR-A-ALMERIA)
```

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