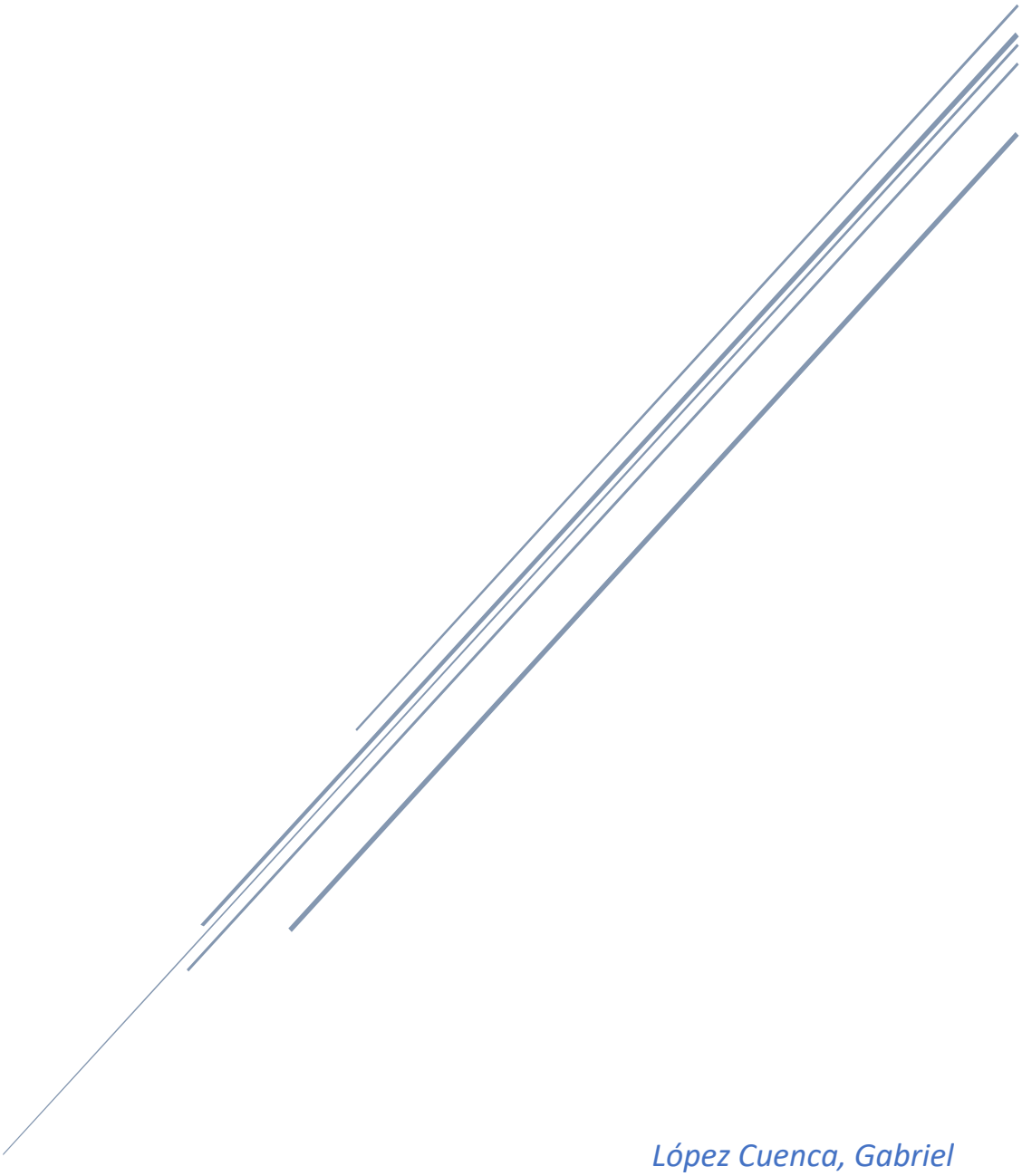
PRÁCTICA 1 LABORATORIO

INTELIGENCIA ARTIFICIAL

Universidad de Alcalá



López Cuenca, Gabriel
Sanz Sacristán, Sergio
Zamorano Ortega, Álvaro

Índice

FORMULACIÓN DEL PROBLEMA 2
Entrada 2
Representación de la información 2
Operadores 2
FUNCIONES AUXILIARES 3
Sucesores_aux 3
Elemento_l 3
Ultimo-lista 3
Poner-final 3
Crear_camino 4
Sucesores 4
Eliminar 4
To-String 5
Camino-final 5
Coste-camino 5
Menor 6
Ordenar 6
MÉTODOS DE BÚSQUEDA 7
Búsqueda por Anchura 7
Búsqueda por Profundidad 7
Búsqueda Optimal 8
EJECUCIÓN 9
Búsqueda 9
Dar_Bienvenida 9

FORMULACIÓN DEL PROBLEMA

Objetivo: realizar un programa en Racket con:

- Entradas: lista de ciudades con la información sobre los enlaces por carretera entre ellas (lista de ciudades contiguas a las que se pueda ir directamente, con indicación de los kilómetros correspondientes), así como una ciudad inicial y otra de destino (meta).
- Salida: ruta para efectuar el viaje junto al coste de recorrerla.

Entrada

- Lista (Lista (A, B…X), Lista (H, J, …, T), …, Lista (L, M, …, Z))
- A,B,…,X representan la distancia a los nodos.

Representación de la información

- Estado inicial: Lista (X) → X ciudad inicial
- X ∈ Naturales
- Estado final: Lista (X, Y, …, Z) → Z ciudad meta
- Y, …, Z ∈Naturales.
- Función esMeta: Lista (X, …, Z)

Operadores

- (X) → ((X, Y), (X, W))

El problema se resolverá mediante los métodos de búsqueda no informada y/o heurísticos, en este caso, mediante búsqueda en anchura, búsqueda en profundidad y búsqueda optimal.

Es importante destacar que la matriz de ciudades que se usa en todas las funciones del programa Racket debe estar constituida por una lista de listas, es decir, una lista con todos los adyacentes desde cada una de las ciudades. Ejemplo:

- Si desde la ciudad 1 se puede ir a la ciudad 2, y, viceversa, la matriz de ciudades tendrá el siguiente aspecto: list (‘(0 1) ‘(1 0)).

FUNCIONES AUXILIARES

Sucesores_aux

- Retorna una lista con el número de las ciudades contiguas de una determinada ciudad. Los parámetros de entrada son la lista de costes de las ciudades adyacentes de la ciudad a evaluar, el propio número de la ciudad y un contador inicializado a 1.

```
(define (sucesores_aux ciudad cont)
 (cond
  [(null? ciudad) '()]
  [(>= (car ciudad) 1) (cons cont (sucesores_aux (cdr ciudad) (+ cont 1)))]
  [else (sucesores_aux (cdr ciudad) (+ cont 1))]))
```

```
> (sucesores_aux '(0 50 98 0 0) 1)
(list 2 3)
```

Elemento_l

- Retorna el elemento de una lista dado su índice y la lista correspondiente.

```
(define (elemento_l lista indice)
 (list-ref lista (- indice 1)))
```

```
> (elemento_l '(0 50 98 0 0) 2)
50
```

Ultimo-lista

- Retorna el último elemento de una lista dada una cierta lista.

```
(define (ultimo-lista lista)
 (elemento_l lista (length lista)))
```

```
> (ultimo-lista '(0 50 98 0 1))
1
```

Poner-final

- Añade el elemento x al final de la lista pasada como l.

```
(define (poner-final x l)
 (reverse (cons x (reverse l))))
```

```
> (poner-final 2 '(0 40 4))
(list 0 40 4 2)
```

Crear_camino

- Dadas dos listas, retorna una lista formada por las listas resultantes de unir la primera de ellas con cada uno de los elementos de la segunda.

```
(define (crear_camino lista_original lista_sucesores)
 (cond
  [(null? lista_sucesores) '()]
  [else (cons (poner-final (car lista_sucesores) lista_original) (crear_camino lista_original (cdr lista_sucesores)))]))
```

```
> (crear_camino '(1 2) '(3 4))
(list (list 1 2 3) (list 1 2 4))
```

Sucesores

- A partir de las funciones anteriores, dada una lista con el camino recorrido hasta el momento, devuelve la lista formada por las listas de los caminos para llegar a cada uno de los sucesores del último nodo del camino original.

```
(define (sucesores lista matriz_ciudades)
 (crear_camino lista (sucesores_aux (elemento_l matriz_ciudades (ultimo-lista lista)) 1)))
```

```
> (sucesores '(1 2) (list '(0 1 1 0 0) '(1 0 0 1 1) '(1 0 0 1 0) '(0 1 1 0 1) '(0 1 0 1 0)))
(list (list 1 2 1) (list 1 2 4) (list 1 2 5))
```

Eliminar

- Pasados un elemento y una lista, devuelve una lista sin el elemento dicho.

```
(define (eliminar n lista)
 (cond
  [(empty? lista) empty]
  [(equal? n (car lista)) (cdr lista)]
  [else (cons (car lista) (eliminar n (cdr lista)))]))
```

```
> (eliminar 2 '(1 2 3))
(list 1 3)
```

To-String

- Convierte en string el camino pasado de acuerdo con el siguiente formato:
  - Los primeros elementos, menos el último, son el propio camino.
  - El último elemento es el coste de dicho elemento.

```
(define (to-string camino)
 (cond
  [(= (length camino) 1) (string-append "-> " (string-append "Coste: " (number->string (car camino))))]
  [else (string-append (string-append (number->string (car camino)) " ") (to-string (cdr camino)))]))
```

```
> (to-string '(1 2 1))
"1 2 -> Coste: 1"
```

Camino-final

- Añade al final del camino para llegar a la ciudad meta el coste de recorrer dicho camino. Para ello le es necesario el camino y la matriz de ciudades con el fin de calcular su coste.

```
(define (camino-final camino matriz_ciudades)
 (cond
  [(empty? camino) '()]
  [else (poner-final (coste-camino camino matriz_ciudades) camino)]))
```

```
> (camino-final '(1 2) (list '(0 1 1 0 0) '(1 0 0 1 1) '(1 0 0 1 0) '(0 1 1 0 1) '(0 1 0 1 0)))
(list 1 2 1)
```

Coste-camino

- Dado un camino y la matriz de adyacencia con los costes entre ciudades devuelve el coste del camino. Para ello del primer elemento del camino obtiene el coste de

llegar al segundo de ellos repetidas veces (hasta que el camino solo tenga un elemento).

```
(define (coste-camino camino matriz_ciudades)
 (cond
  [(= (length camino) 1) 0]
  [else (+ (elemento_l (elemento_l matriz_ciudades (elemento_l camino 1)) (elemento_l camino 2))(coste-camino (cdr camino) matriz_ciudades))]))
```

```
> (coste-camino '(1 2) (list '(0 1 1 0 0) '(1 0 0 1 1) '(1 0 0 1 0) '(0 1 1 0 1) '(0 1 0 1 0)))
1
```

Menor

- Dada una lista de listas (caminos) devuelve el menor de ellos, es decir, el que menor coste suponga recorrerlo. Si el coste es el mismo en ambos de ellos, devuelve el que más profundidad presente en la lista.

```
(define (menor lista matriz_ciudades)
 (cond
  [(empty? lista) empty]
  [(empty? (cdr lista)) (car lista)]
  [(< (coste-camino (car lista) matriz_ciudades) (coste-camino (car (cdr lista)) matriz_ciudades)) (menor (cons (car lista) (cddr lista)) matriz_ciudades)]
  [else (menor (cdr lista) matriz_ciudades)]))
```

```
> (menor (list '(1 2) '(1 2 3 4 5)) (list '(0 1 1 0 0) '(1 0 0 1 1) '(1 0 0 1 0) '(0 1 1 0 1) '(0 1 0 1 0)))
(list 1 2)
```

Ordenar

- Ordena una lista de caminos de menor a mayor, es decir, de forma creciente respecto a los costes de los caminos que la forman.

```
(define (ordenar lista matriz_ciudades)
 (cond
  [(empty? lista) empty]
  [else (cons (menor lista matriz_ciudades) (ordenar (eliminar (menor lista matriz_ciudades) lista) matriz_ciudades))]))
```

```
> (ordenar (list '(1 2) '(1 2 3 4 5)) (list '(0 1 1 0 0) '(1 0 0 1 1) '(1 0 0 1 0) '(0 1 1 0 1) '(0 1 0 1 0)))
(list (list 1 2) (list 1 2 3 4 5))
```

MÉTODOS DE BÚSQUEDA

Búsqueda por Anchura

Para la práctica hemos desarrollado varios métodos de búsqueda, comenzando por el método de búsqueda por anchura. Para este método se ha utilizado el siguiente código Racket:

```
(define (busqueda_a abiertos cerrados meta matriz_ciudades)
 (when (not (empty? abiertos))
  (let ([actual (car abiertos)])
   (cond
    [(equal? (ultimo-lista actual) meta) actual]
    [(member (ultimo-lista actual) cerrados) (busqueda_a (cdr abiertos) cerrados meta matriz_ciudades)]
    [else (busqueda_a
           (append (cdr abiertos) (sucesores actual matriz_ciudades))
           (cons (ultimo-lista actual) cerrados) meta matriz_ciudades)])))))
```

Recibe como parámetros una lista de abiertos, otra de cerrados, la meta buscada y una lista de listas que contiene las conexiones entre las diferentes ciudades.

Realización: mientras la lista de abiertos no está vacía, seleccionamos el último valor de la lista actual y lo comparamos con la meta; si coincide entonces devolvemos la lista actual como el camino encontrado, si este valor pertenece a cerrados entonces llamamos recursivamente a la función sin el valor seleccionado en la lista de abiertos. Si no se cumple ninguno de estos casos, llamamos de nuevo a la función de búsqueda en anchura, introduciéndole como parámetros la lista de abiertos añadiendo en los últimos valores los sucesores del nodo en el que nos encontramos, eliminando de la lista el nodo en el que acabamos de estar, la lista de cerrados añadiendo el nodo actual, la meta y la misma matriz de ciudades.

Búsqueda en Profundidad

Posteriormente hemos realizado el método de búsqueda por profundidad.

Realización: se realiza una búsqueda de forma similar a la de anchura, la diferencia es que si el nodo en el que estamos no es la meta y no está en cerrados entonces al llamar de nuevo a la función de búsqueda en profundidad introducimos como parámetros la lista de abiertos con los sucesores del nodo al principio de ella, eliminando de la lista el nodo en el que estamos, la lista de cerrados añadiendo el nodo que acabamos de estudiar, la meta y la matriz de las ciudades sin modificar.

El código Racket de dicho método es el que sigue:

```
(define (busqueda_p abiertos cerrados meta matriz_ciudades)
 (when (not (empty? abiertos))
  (let ([actual (car abiertos)])
   (cond
    [(equal? (ultimo-lista actual) meta) actual]
    [(member (ultimo-lista actual) cerrados) (busqueda_p (cdr abiertos) cerrados meta matriz_ciudades)]
    [else (busqueda_p
           (append (sucesores actual matriz_ciudades) (cdr abiertos))
           (cons (ultimo-lista actual) cerrados) meta matriz_ciudades)])))
```

Búsqueda Optimal

Por último, se realiza la búsqueda optimal para la que se ha utilizado el siguiente código, donde la función recibe los mismos datos que anteriormente:

```
(define (busqueda_o abiertos cerrados meta matriz_ciudades)
 (when (not (empty? abiertos))
  (let ([actual (car abiertos)])
   (cond
    [(equal? (ultimo-lista actual) meta) actual]
    [(member (ultimo-lista actual) cerrados) (busqueda_o (ordenar (cdr abiertos) matriz_ciudades) cerrados meta matriz_ciudades)]
    [else (busqueda_o
           (ordenar (append (cdr abiertos) (sucesores actual matriz_ciudades)) matriz_ciudades)
           (cons (ultimo-lista actual) cerrados) meta matriz_ciudades)])))
```

Realización: si la lista de abiertos no está vacía, cogemos el último valor de la lista de abiertos y si coincide con la meta terminamos, si no, comprobamos si es miembro de la lista de cerrados, si es así entonces llamamos de nuevo a la función de búsqueda optimal eliminando el nodo actual de la lista de abiertos. Si no se cumplen ninguna de estas condiciones, llamamos de nuevo a la función de búsqueda optimal pasándole como parámetros la lista de abiertos eliminando de ella el nodo actual y añadiéndole los sucesores del nodo en el que nos encontrábamos, y ordenamos la lista de abiertos de menor a mayor de acuerdo al coste de los caminos que en ella se encuentran. Además, introducimos como parámetros la lista de cerrados añadiendo el nodo que se está estudiando actualmente, la meta y la matriz de ciudades con los datos de costes y adyacencia correspondientes.

EJECUCIÓN

Búsqueda

- Para iniciar la ejecución es necesario invocar a la función búsqueda en la que se comprueba que la meta se encuentre dentro de las posibilidades incluidas en la matriz de adyacencia, y una vez aceptada dicha condición, se pasa a ejecutar uno de los métodos de búsqueda desarrollados en función del parámetro introducido. Finalmente, presentará el camino, en caso de existir, para ir desde el nodo inicial al meta. Recibe como parámetros:
  - Matriz de adyacencia entre ciudades.
  - Tipo de búsqueda a realizar (1,2,3).
  - Nodo inicial.
  - Nodo meta.

```
(define (busqueda lista_ciudades tipo inicial meta)
 (if (> meta (length lista_ciudades)) (display "  *Meta no valida*\n\n")
   (cond
    [(= tipo 1) (display (string-append "  ** Camino: "
          (string-append (to-string (camino-final (busqueda_a (list(list inicial)) empty meta lista_ciudades) lista_ciudades)) " **\n\n")))]
    [(= tipo 2) (display (string-append "  ** Camino: "
          (string-append (to-string (camino-final (busqueda_p (list(list inicial)) empty meta lista_ciudades) lista_ciudades)) " **\n\n")))]
    [(= tipo 3) (display (string-append "  ** Camino: "
          (string-append (to-string (camino-final (busqueda_o (list(list inicial)) empty meta lista_ciudades) lista_ciudades)) " **\n\n")))]
    [(= tipo 4) (display (string-append "  ** Camino: "
          (string-append (to-string (camino-final (busqueda_ap (list(list inicial)) empty meta lista_ciudades) lista_ciudades)) " **\n\n")))]
    [else (display "Metodo de busqueda no valido\n")])))
```

Dar_Bienvenida

- Únicamente mostrará una guía de cómo ejecutar el programa.

```
(define (dar_bienvenida)
 (display "\nBIENVENIDO A LA BUSQUEDA DE RUTAS\n ->Introduzca:\n  (busqueda *Matriz de conexiones* *Tipo de busqueda(Anchura -> 1, Profundidad -> 2, Optimal -> 3, A* -> 4)* *Estado inicial* *Estado meta*)\n\n"))
```

```
BIENVENIDO A LA BUSQUEDA DE RUTAS
 ->Introduzca:
  (busqueda *Matriz de conexiones* *Tipo de busqueda(Anchura -> 1, Profundidad -> 2, Optimal -> 3)* *Estado inicial* *Estado meta*)
```