Universidad de Alcalá

Escuela Politécnica Superior

Universidad de Alcalá

PECL1

Búsqueda del camino mínimo en un grafo

Inteligencia Artificial

Laboratorio Miércoles 12:00 – 14:00

Grado en Ingeniería Informática – Curso 2018/2019

Eduardo Graván Serrano – 03212337L

Marcos Barranquero Fernández – 51129104N

Adrián Montesinos González – 51139629A

DISCUSIÓN Y FORMULACIÓN DEL PROBLEMA

Este primer trabajo de laboratorio pide realizar un programa en racket que sea capaz de, aplicando un algoritmo de búsqueda de los vistos en las clases de teoría, encontrar el camino mínimo entre dos nodos de un grafo. En el enunciado de la práctica se establece que los nodos de los grafos son ciudades, mientras que las aristas serían las carreteras que unen estas ciudades, aunque realmente se puede el programa desarrollado para encontrar el camino gráfico aplicado a cualquier grafo.

Se pide también que el programa lea el grafo desde un fichero de texto, en el cual se especifica tanto el nodo origen como el nodo destino, así como las distintas aristas o carreteras y los costes de recorrer estas aristas. El formato del archivo de texto es el siguiente:

```
Origen:
Oviedo
Destino:
Murcia
(Oviedo Bilbao 304)
(Bilbao Valladolid 280)
(Bilbao Madrid 395)
(Bilbao Zaragoza 324)
```

Es necesario seguir este formato a la hora de crear el .txt o el programa no será capaz de leer el archivo correctamente. La captura presentada no hace referencia a un grafo completo, solo se usa para mostrar el formato adoptado.

PLANTEAMIENTO DE LA SOLUCIÓN

Tenemos por lo tanto un grafo que indica los caminos posibles entre ciudades y las distancias entre estas ciudades. Hemos implementado tres métodos de búsqueda de los vistos en clase, que son:

- **Búsqueda optimal:** Para ella, ordenaremos la lista de abiertos de menor a mayor en función de las distancias a las ciudades. Se cogerá el primero de la lista de abiertos, es decir, el siguiente nodo será la ciudad con menor distancia al nodo actual, y se seguirá estudiando desde ahí hasta que lleguemos al nodo destino.
- **Búsqueda en anchura:** No se tiene en cuenta la distancia, simplemente se exploran todas las posibles ciudades a las que podemos ir desde el nodo actual y, si ninguna de ellas es el nodo destino, se exploran los sucesores de estos nodos. Es decir, los sucesores de los nodos se añaden en la cola de la lista de abiertos.
- **Búsqueda en profundidad:** Tampoco se tiene en cuenta la distancia, se explora buscando en profundidad desde una de las ramas del árbol de búsqueda. Si no se encuentra la solución, se exploran las otras ramas. Es decir, los sucesores de los nodos se añaden en la cabeza de la lista de abiertos.

Debido a que los grafos no son muy grandes, la búsqueda en anchura es posible y encontrará siempre una solución óptima (sin mirar las distancias de los caminos).

La búsqueda en profundidad también encontrará solución ya que no hay ramas infinitas en el grafo, y encontrará también una solución óptima (sin mirar las distancias de los caminos).

La búsqueda optimal será la única que encontrará una solución óptima de verdad, ya que mira las distancias entre los nodos y toma decisiones en base a ello. Siempre encontrará solución y esta será realmente óptima.

FUNCIONES DEL PROGRAMA

ESTRUCTURAS

Contamos con dos estructuras principales:

- Arista: Estructura conformada por dos ciudades unidas por un camino, y su coste.
- Estado: estructura formada por el camino de ciudades recorridas y el kilometraje acumulado.

LECTURA DEL FICHERO

Para leer el fichero, se añaden todos los elementos del fichero transformados a las estructuras convenientes en una lista. Después, de esta lista, se extrae origen, destino, y el resto de las aristas que conforman el grafo.

Al leer el fichero, lo primero que se debe leer es el estado origen y la ciudad objetivo.

Para el estado origen se crea un nuevo estado, con kilometraje acumulado a 0 y la ciudad inicial.

El destino se almacena como una String con el nombre de la ciudad objetivo.

Para leer las aristas, se leen línea a línea el fichero. Cada línea es una arista que es leída en forma de cadena de símbolos. Es por ello que se tiene una función que transforma la línea leída en una arista.

Mientas no se encuentre un EOF, se seguirá leyendo recursivamente el fichero y concatenando aristas a la lista de elementos leídos.

FUNCIONES AUXILIARES PARA ALGORITMOS

SUCESORES

Sucesores es una función que, dado un grafo y un estado, devuelve los sucesores de dicho estado. Estos son, las posibles ciudades a las que ir desde la última ciudad visitada, añadiendo su kilometraje al kilometraje acumulado del estado. Su funcionamiento es el siguiente:

1. Si el grafo está vacío, devuelve lista vacía.
2. Si la arista del grafo contiene el estado inicial, o una ciudad previamente visitada como destino de esa arista, seguir buscando.
3. Si en esa arista del grafo se encuentra la ciudad en la que se está como origen, devolverla como una lista y concatenar con la llamada recursiva.

COMPARADOR, MÍNIMO Y MÍNIMO AUX

Estas funciones son utilizadas para el algoritmo optimal.

- Mínimo: devuelve el mínimo de la lista de abiertos, haciendo uso de mínimo-aux.
- Mínimo-aux: busca y devuelve el mínimo de la lista de abiertos, recorriendo dicha lista recursivamente.
- Comparador: permite compara el kilometraje de un estado con otro. Devuelve el que tenga menor kilometraje.

CONTENIDO COMÚN EN TODOS LOS ALGORITMOS

Todos los algoritmos realizan la siguiente funcionalidad:

1. Se llama a la función de búsqueda con el estado inicial donde se irá acumulando el camino, y una lista vacía, que simboliza la lista de abiertos.
2. Si el primer elemento de abiertos es el destino, se imprime el recorrido realizado y el coste, y se finaliza el algoritmo.
3. Si no:
    a. Se imprime el estado actual
    b. Se extraen los sucesores del estado actual y se concatenan con abiertos.
    c. Se llama recursivamente pasando el siguiente estado de abiertos como estado actual y la lista de abiertos restante.

La diferencia entre los algoritmos es cómo se tratan los sucesores y la lista de abiertos en dicha llamada recursiva.

- **Búsqueda en anchura:** la lista de abiertos consiste en una lista FIFO. Al extraer sucesores, la nueva lista de abiertos es la resultante de concatenar abiertos + sucesores.
- **Búsqueda en profundidad:** la lista de abiertos consiste en una lista LIFO. Al extraer sucesores, la nueva lista de abiertos es la resultante de concatenar sucesores+abiertos.
- **Búsqueda optimal:** la lista de abiertos consiste en una cola de prioridad. Al extraer sucesores, la nueva lista de abiertos es la resultante de concatenar abiertos + sucesores y pasarla por una función auxiliar que ordena los estados de dicha lista de menor a mayor coste.

FUNCIONES AUXILIARES DE IMPRESIÓN

Se tienen funciones para imprimir un estado y una arista. También podemos, haciendo uso de estas, imprimir un camino, que es una lista de estados, y un grafo, que es una lista de aristas.

También disponemos de una función que imprime una lista invertida, esto es debido a que la lista de estados se almacena de último visitado a primero visitado, y para imprimir en el algoritmo queremos que sea de primero visitado a último, como se llevaría una lista de ciudades visitadas en la vida real.

ANEXO – CÓDIGO

```
#lang racket

; USAR IN PARA LEER LA ENTRADA DE TEXTO y poner en el archivo txt "destino" "100" etc.
;

; ----- Estructuras ------- ;
; Estado: estado en el mapa de estados. (Que no el de ciudades) {camino seguido, coste
en km}
(define-struct estado (camino km))
; Arista: unión de una ciudad a otra {ciudadA, ciudadB, coste}
(define-struct arista (origen destino km))


; ----- Lectura de fichero  -------- ;

(define (transforma-arista lista-leida)
; Dada una línea del archivo de texto, la transforma en dos aristas:
  (list
  ; De destino a origen
  (make-arista
    (symbol->string (car lista-leida))  ; Ciudad A
    (symbol->string (cadr lista-leida)) ; Ciudad B
    (caddr lista-leida)                 ; Coste
  )
  ; De origen a destino
  (make-arista
    (symbol->string (cadr lista-leida))  ; Ciudad A
    (symbol->string (car lista-leida)) ; Ciudad B
    (caddr lista-leida)                 ; Coste
  )
  )
)

(define (leer-grafo-aux entrada)
; Lee el archivo de grafo recursivamente
  (let ([posible-arista (read entrada)])
    (cond
      ; Si he llegado a final de línea, termino:
      [(eof-object? posible-arista) (list)]
      ; Si la siguiente línea es el origen o destino
      ; lo tomo y llamo recursivamente.
      [(or (equal? posible-arista 'Origen:) (equal? posible-arista 'Destino:))
        (append
          (list (symbol->string (read entrada)))
          (leer-grafo-aux entrada))]
```

```
      ; Si no, creo la arista y llamo recursivamente
      [(append (transforma-arista posible-arista) (leer-grafo-aux entrada))]
    )
  )
)

(define (leer-grafo)
  (define entrada (open-input-file "grafo-ciudades.txt"))
  (leer-grafo-aux entrada)
  ;(close-input-port entrada)
)

; ------- Variables globales ------ ;

(define estado-inicial (make-estado (list (car (leer-grafo))) 0))
(define objetivo (cadr (leer-grafo)))
(define grafo (cddr (leer-grafo)))


; ------- Algoritmo de búsqueda ------ ;


;Devuelve los sucesores de un estado, dado un grafo de ciudades
(define (sucesores grafo estado)
  (cond
    ; Si está vacío, devuelve vacío
    [(empty? grafo) empty]
    ; Elif
    [(or
      ; Si no son iguales el origen y el elemento
      (not (equal? (arista-origen (car grafo)) (car (estado-camino estado))))
      ;O No está el destino en el camino recorrido
      (member (arista-destino (car grafo)) (estado-camino estado)))
     ; Seguir buscando el destino en el resto del camino recorrido
              (sucesores (cdr grafo) estado)]
    [else (cons
           ; Cabeza
           (make-estado
            ; Nuevo camino
            (cons
             ;Cabeza: cabeza del camino (nuevo nodo)
             (arista-destino (car grafo))
             ; Cola: lista que ya teniamos
             (estado-camino estado))
            ; distancia en km
            (+ (arista-km (car grafo)) (estado-km estado)))
           ; Cola
          (sucesores (cdr grafo) estado))]
    )
```

```
  )


; Devuelve el minimo en la lista de estados
(define (minimo-aux colaestados min)
  (cond
    [(null? colaestados) min]
    [(comparador (car colaestados) min)
     (minimo-aux (cdr colaestados) (car colaestados))]
    [else
     (minimo-aux (cdr colaestados) min)]
    )
  )

 ; Comparador de dos elementos del struct por la distancia en kilometros, devuelve true
si el primero es menor que el segundo
(define (comparador elemento1 elemento2)
  (< (estado-km elemento1) (estado-km elemento2))
)

; Esto pilla el minimo de la lista de estados dada
(define (minimo listaestados)
  (if (null? listaestados)
      #f
      (minimo-aux (cdr listaestados) (car listaestados))))

; Genera la lista ordenada de sucesores
(define (cola-de-prioridad lista-sucesores)
  (cond
    [(equal? (length lista-sucesores) 1) car lista-sucesores]
    [else
     (let ([estado-minimo (minimo lista-sucesores)])
           (cons  estado-minimo (cola-de-prioridad (remove estado-minimo lista-
sucesores))))]))


; ------- Funciones principales ------- ;

(define (anchura estado-actual abiertos)
  ; Abiertos es una cola FIFO. ( Abiertos + sucesores)
  (cond
    ; Si el estado actual es el objetivo, imprimo y pa casa
    [(equal? (car(estado-camino estado-actual)) objetivo)
     (display "\n¡Has llegado a tu destino!\n")
     (imprime-estado estado-actual)]
    ; Si no, pillo sucesor y tiro palante
    [
     ; Imprimo estado
     (display "\nEstado: ")(imprime-estado estado-actual)
```

```
     ; Llamo a main-aux tomando el estado más barato y pasando el resto como abiertos
     (anchura (car (append abiertos (sucesores grafo estado-actual)))
           (cdr (append abiertos (sucesores grafo estado-actual))))
    ]
  )
)

(define (profundidad estado-actual abiertos)
  ; Abiertos es una cola LIFO. (Sucesores + Abiertos)
  (cond
    ; Si el estado actual es el objetivo, imprimo y pa casa
    [(equal? (car(estado-camino estado-actual)) objetivo)
     (display "\n¡Has llegado a tu destino!\n")
     (imprime-estado estado-actual)]
    ; Si no, pillo sucesor y tiro palante
    [
     ; Imprimo estado
     (display "\nEstado: ")(imprime-estado estado-actual)
     ; Llamo a main-aux tomando el estado más barato y pasando el resto como abiertos
     (profundidad (car (append (sucesores grafo estado-actual) abiertos))
           (cdr (append (sucesores grafo estado-actual) abiertos)))
    ]
  )
)

(define (optimal estado-actual abiertos)
  ; Abiertos es una cola de prioridad ( menor (sucesores+abiertos))
  (cond
    ; Si el estado actual es el objetivo, imprimo y pa casa
    [(equal? (car(estado-camino estado-actual)) objetivo)
     (display "\n¡Has llegado a tu destino!\n")
     (imprime-estado estado-actual)]
    ; Si no, pillo sucesor y tiro palante
    [
     ; Imprimo estado
     (display "\nEstado: ")(imprime-estado estado-actual)
     ; Llamo a main-aux tomando el estado más barato y pasando el resto como abiertos
     (optimal (car (cola-de-prioridad (append abiertos (sucesores grafo estado-
actual))))
           (cdr (cola-de-prioridad (append abiertos (sucesores grafo estado-actual)))))
    ]
  )
)

(define (main)
  (display "\nAlgoritmo optimal (cola de prioridad): \n")
  (optimal estado-inicial (list ))
  (display "\n\nAlgoritmo búsqueda en profundiad (cola LIFO): \n")
  (profundidad estado-inicial (list ))
```

```
  (display "\n\nAlgoritmo profundidad (cola FIFO): \n")
  (anchura estado-inicial (list ))
)

; ------- Funciones para imprimir cosas ------ ;

; Imprime un estado
(define (imprime-estado estado)
  (display "Ruta: ")(display "( ")(pintar-lista-invertida (estado-camino estado))
(display ")")
                           (display ", Coste: ") (display (estado-km estado)))

; Imprime la lista de caminos asociada a un estado invertida
(define (pintar-lista-invertida lista)
  (cond
    [(null? (cdr lista)) (display(car lista)) (display " ")]
    [else (pintar-lista-invertida (cdr lista))(display (car lista))(display " ")]
    )
  )


; Imprime una lista de estados
(define (imprime-estados lista-estados)
  (for-each imprime-estado lista-estados)
)

; Imprime una arista
(define (imprime-arista arista)
  (display (list (arista-origen arista) (arista-destino arista) (arista-km arista))))

; Imprime un grafo
(define (imprime-grafo grafo)
  (cond
    [(empty? grafo)(display "")]
    [(imprime-arista (car grafo)) (display ",") (imprime-grafo (cdr grafo))]
  )
)
```