Memoria P1

Inteligencia Artificial

Grupo 2361

Autores: Guillermo Rodríguez
y Javier Muñoz

Ejercicio 1: Distancia Coseno (1.5 ptos)

1.1) Para este ejercicio lo que se nos pide es calcular la distancia coseno entre dos vectores. Para ello lo que hacemos es adaptar la fórmula que se nos da en el enunciado:

$$\text{cosine-distance}(\mathbf{x},\mathbf{y}) = 1 - \cos(\theta) = 1 - \frac{\mathbf{x}\cdot\mathbf{y}}{\|\mathbf{x}\|_2\|\mathbf{y}\|_2} = 1 - \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$$

Para ello lo que hicimos fue dividir el calculo en dos partes, la parte de arriba y la parte abajo, usando funciones auxiliares, las cuales hacían operaciones recursivas recorriendo los elementos de la lista, y aplicando dicha fórmula sobre cada uno de los elementos.

Una vez que la función estaba hecha, tuvimos que hacer la segunda función usando mapcar, usando exactamente la misma fórmula, pero codificándola de una forma diferente.

- (cosine-distance ’(1 2) ’(1 2 3)) -> 0.40238577
- (cosine-distance nil ’(1 2 3)) -> - /: division by zero
- (cosine-distance ’() ’()) -> - /: division by zero
- (cosine-distance ’(0 0) ’(0 0)) : He aquí nuestro dilema, matemáticamente siguiendo la fórmula obtendríamos que la fórmula acabaría en el calculo de 1 - 0/0, dado que sabemos que el ángulo que se forma entre dos vectores que son iguales es 0, y que cos(0º) = 1, la distancia coseno sería de 1-1 = 0, no sabemos si tenemos que demostrarlo programándolo o si por conocimientos básicos de matemáticas deberíamos de llegar a esta conclusión en el cálculo.

1.2) En este apartado, lo que se nos pedía era, viendo los valores que nos devolvía las funciones cosine-distance-rec, si este valor sobrepasaba un nivel de confianza, concatenarlo para añadirlo a la lista de resultados.

Resultados:

(order-vectors-cosine-distance ’(1 2 3) ’()): Retorna nil ya que entra en el caso base de la recursion donde la lista de vectores esta vacia.

(order-vectors-cosine-distance ’() ’((4 3 2) (1 2 3))): Nos da error debido a una division por 0.

1.3) En este ejercicio se nos pide la implementación de la función get-vectors-category, la cual clasificaba los vectores por su distancia coseno. La idea era recibir un conjunto de categorías, cuyo primer elemento es su identificador, un conjunto de vectores que representan “textos”, y cuyo primer elemento es

su identificador y devolviese una lista con pares de identificador-distancia.

1.4) La salida de las diferentes entradas son:

(get-vectors-categories '(()) '(()) #'cosine-distance) -> - /: division by zero

(get-vectors-categories '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance) -> NIL is not a number

Ejercicio 2: Raíces de una función (1.5 ptos)

2.1) Para este ejercicio teníamos que usar un método de aproximación para encontrar las raíces de una función (valores donde la función toma el valor y=0)
Se nos proporcionaba la siguiente fórmula:

$$f(r) = f(x_0 + h) \approx f(x_0) + hf'(x_0) \implies h \approx -\frac{f(x)}{f'(x)}.$$

Por tanto,

$$r = x_0 + h \approx r = x_0 - \frac{f(x)}{f'(x)}.$$

Teniendo esto en cuenta, las nuevas estimaciones x_{n+1} de r se generan a partir del valor anterior x_n:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, n = 0, 1, 2...$$

El método newton es aplicado a la función de lisp newton, usando una función auxiliar llamada calcular para repartir el trabajo de forma equivalente entre ambos.

2.2) Posteriormente one-root-newton calcula una raíz de la función pasándole dicha función, su derivada, una lista de semillas (valores de donde empezar a buscar) y un número concreto de iteraciones. La fórmula devuelve NIL si no encuentra nada con las iteraciones dadas y la primera raíz que encuentra si es así.

2.3) Por último, la función all-roots-newton, la cual haya no solo una, sino todas las raíces de una función, hace llamadas recursivas así misma y a la función definida previamente, newton, y concatena los resultados que ésta le va devolviendo.

Ejercicio 3: Combinación de listas (1 pto)

3.1) Para resolver este apartado simplemente implementamos la función combine-elt-lst() llamando a cons con la lista formada con el

elemento y el primero de la lista y la llamada recursiva a la propia función con el elemento y el resto de la lista.
Cuestiones:

1. **(combine-elt-lst 'a nil):** El retorno de esta llamada es nil. Este resultado se debe a que la función se mete en el caso base, donde la lista (parámetro recursivo) esta vacia y se retorna nil.
2. **(combine-elt-lst nil nil):** También retorna nil por el misma razon que en el anterior apartado.
3. **(combine-elt-lst nil '(a b)):** En este caso la salida seria ((NIL A) (NIL B)). Donde se combina el elemento nulo con los elementos de la lista.

3.2) En este apartado implementamos la función **combine-lst-lst().** En la implementación llamamos a combine-elt-lst fijando lst a una de las listas y recorriendo la otra lista usando sus elementos como elt.

Cuestiones:

1. **(combine-lst-lst nil nil):** La salida seria nil, ya que entraría en el caso base de una de las listas vacías. Tiene sentido ya que la combinación de dos listas vacias seria otra lista vacia.
2. **(combine-lst-lst '(a b c) nil):** De nuevo la salida es nil. En nuestro caso la función llamaría a recursivamente a combine-elt-lst usando como lista fija la segunda. Al entrar en esta función, nos meteríamos en su caso base (lista vacía) y se estaría llamando a append de nil recursivamente.
3. **(combine-lst-lst nil '(a b c)):** También retornaría nil. En este caso porque entra en el caso base de la funcion principal, que comprueba que la primera lista este vacía, y retornaría nil.

3.3) Esta seria la función mas compleja a nuestro parecer del este ejercicio. Lo que hacemos es llamar recursivamente a combine-lst-lst de manera que se combinan todas las listas. También utilizamos la función mapcar para resolver un problema que tuvimos con listas anidadas.
Cuestiones:

1. **(combine-list-of-lsts '(() (+) (1 2 3 4))):** La salida es nil ya que desde el principio llama a combine-lst-lst con la primera lista vacía, entrando en el caso base.
2. **(combine-list-of-lsts '((a b c) () (1 2 3 4))):** Retorna nil. Como en el apartado 3.2.2, al combinar las dos primeras listas, la lista a retornar que se va actualizando llega a nil, y desde ese momento siempre lo sera por lo explicado en el aparado 3.2
3. **(combine-list-of-lsts '((a b c) (1 2 3 4) ())):** La salida es nil. En este caso lo la lista resultante era una combinación de las dos primeras listas, pero al combinarse con la ultima vacía, se actualizaria y retornaría nil.
4. **(combine-list-of-lsts '((1 2 3 4))):** En este caso la salida seria ((1) (2) (3) (4)). Podríamos haber esperado un nil, ya que

llamaría a combine-lst-lst de (1 2 3 4) con la siguiente lista de la lstoflsts pero debido a la llamada a mapcar, el resultado queda así.

5. **(combine-list-of-lsts ’(nil)):** Retorna nil. La función detecta una lista (no sabe que dentro solo tiene el elemento nil) y llama a combine-lst-lst con nil como parámetro y se mete en su caso base.
6. **(combine-list-of-lsts nil):** La salida es (nil). La funcion entra directamente en su caso base y retorna (nil).

Ejercicio 4: Arboles de verdad en lógica proposicional (5 ptos)

Nosotros no utilizamos funciones para implementar las reglas. Tenemos toda la implementación en 4 funciones:

truth-tree: Esta es la función principal. Simplemente crea una lista vacía que le pasara a la función expand-truth-tree, a la que aplicara la función tiene-contradicciones y en caso de que no las tenga retornará TRUE. En su defecto retornara nil.

expand-truth-tree: Esta función es una función recursiva, que recibe una lista (final) que ira actualizando y sera la que acabara retornando, y una expresión lógica. Dependiendo de en que parte de la expresión estemos analizando, se ira parando en distintos ifs, de manera que si detecta un condicional o un bicondicional, traducirá la expresión a una mas profunda con literales y operaciones or y and. En el caso de detectar una or, la función añadirá a la lista final todos los literales de la disyunción. En el caso de una and, meterá todos los literales de la conjunción en forma de lista, para darnos cuenta que solo es valida si todas se cumplen. Por ultimo, si esta función recibe un literal como expresión, lo retorna.

Flow: esta es una función auxiliar que llamara la función anterior al detectar una operación n-aria. Su función es retornar la lista de las salidas de las llamadas a expand-truth-tree de los literales de las operaciones n-arias.

Tiene-contradicciones: esta función es la encargada de detectar contradicciones en una lista de literales. Recibe la lista y comprobara que cada literal no tenga su contrario en

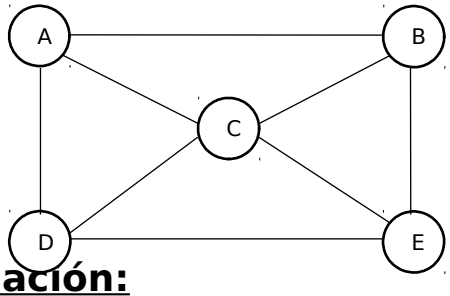
los siguientes de la lista. Barrerá la lista de manera recursiva avanzando, si llega al final, no existen contradicciones y retorna TRUE.

(* Este ejercicio no funciona completamente, ya que a la hora de meterse en operaciones n-arias, si los operandos son expresiones en vez de literales puros, al expandirlos retornan mal los paréntesis. Ademas, la funcion de tiene-contradicciones no esta completada la parte de las subramas ya que esperamos a completar el expand-truth-tree para terminarla, pero al no completar la función de expandir, no pudimos).

Ejercicio 5: Búsqueda en anchura (1 pto)

5.1)

- Grafos especiales

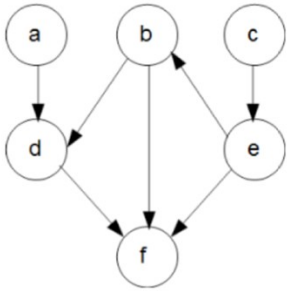


Explicación:

En este caso el Algoritmo de búsqueda en anchura es tiene los siguientes pasos:

Si nuestra búsqueda comienza desde A, primero descubrirá los nodos más cercanos que tiene, en nuestro caso serían B, C y D, después visitará B, y empieza a descubrir los nodos que no están descubiertos todavía, en este caso E, posteriormente visita C y busca nodos que todavía no están descubiertos, como el único al que tiene acceso es E y ya está visitado, C es cerrado, por último en este nivel visitamos D, buscamos nuevos nodos, al no encontrar nada, lo cerramos, lo que hacemos en pasar a E. Una vez en E, buscamos nuevos nodos. Al no encontrar ninguno, cerramos E y terminamos de buscar nodos.

- Caso típico (grafo dirigido)

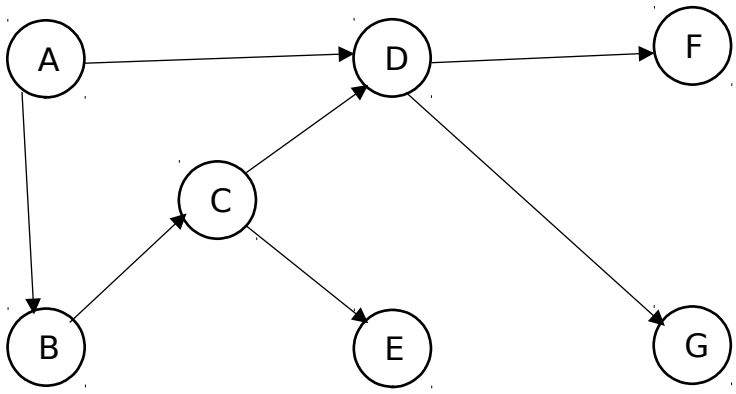


Explicación:

Para este grafo (el de el enunciado) el cual es dirigido el algoritmo varía un poco del que se usan en grafos no dirigidos.

Empezamos por ejemplo en C, y de ahí descubrimos los nodos a los que se pueden acceder, en este caso E, una vez que E es visitado, descubrimos los nodos a los que se pueden acceder desde él, que son B y F, visitamos B por ejemplo. Descubrimos los nodos que son accesibles desde B, en este caso D, el cual marcamos como descubierto. Luego vamos a F, que está al mismo nivel que B, lo marcamos como visitado y descubrimos los nodos accesibles desde él, en este caso ninguno. Ahora con marcamos D como visitado e intentamos acceder a algún nodo. Al no encontrar ninguno, buscamos otro nodo que no esté visitado, en este caso A, lo visitamos e intentamos descubrir otros nodos. Al no haber mas nodos que descubrir nuestra búsqueda a finalizado.

- Casi típico distinto al anterior



Explicación:

En este grafo empezaremos a recorrerlo desde A, marcando visitado como A. Una vez que hemos visitado A, descubrimos los nodos adyacentes, que son B y D. Visitamos B y descubrimos los nodos adyacentes, en este caso C. Por otra parte, tenemos D, de los cuales descubrimos los nodos F y G. Pasamos a C, visitándolo, y descubriendo los nodos a los que podemos acceder, en este caso E. Luego pasamos a los dos nodos descubiertos desde D, que son F y G, primero con F lo visitamos e intentamos descubrir mas nodos, a no encontrarlo vamos a G, lo visitamos e intentamos explorar mas nodos, al no encontrar ninguno pasamos a E. Marcamos E como visitado, al no poder descubrir mas nodos, terminamos la BFS.

5.2) Escribir un algoritmo correspondiente al BFS.

El pseudocódigo de BFS podría explicarse como:

Sea un Grafo G(V, E), podemos representar el BFS como:

```
BFS(G, v)

        Marcar(v)
                Poner (v) en una COLA
        Mientras la COLA no sea vacía hacer
                                Quitar el primer elemento w de la COLA
                Para cada vértice x adyacente a w hacer
                        si x no esta marcado
                                entonces marcar x
                                poner x en la COLA
```

Fuente: https://www.cs.us.es/cursos/cc-2009/material/bfs.pdf

5.3) Una vez que entendemos como ha funcionado, pasamos a pegar el código y entenderlo para el apartado 4.

5.4) Comentamos el código que se nos ha sido proporcionado.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;; Breadth-first-search in graphs
;;;
(defun bfs (end queue net)
  ;Si la cola se ha quedado vacía, terminamos
  (if (null queue) '()
    ;Si no, guardamos los valores del path (recorrido) y node
    (let* ((path (first queue))
      (node (first path)))
    ;;; si tenemos que el nodo actual es el mismo que el nodo que queremos encontrar
    ;;; lo que haremos será devolver el path (lista acumulada de nodos), del reverso, ya que
    ;;; tal y como vamos guardando los nodos que encontramos es al revés del orden natural.
    (if (eql node end)
      (reverse path)

    ;;; Si no encontramos el nodo del final, repetimos la operación, uniendo el resto de la cola con
    ;;; los nuevos nodos descubiertos, de tal manera que recursivamente vayamos explorando el grafo
    (bfs end
      (append (rest queue)
        (new-paths path node net))
      net)))))

(defun new-paths (path node net)
  (mapcar #'(lambda(n)
    (cons n path))
  (rest (assoc node net))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.5) El código proporcionado es el siguiente:

```
(defun shortest-path (start end net)
(bfs end (list (list start)) net))
```

Por lo que hemos intuido, este código lo que hace es llamar a la función de bfs principal, la cual realiza todas las operaciones de búsqueda.

Cuando usamos esta función, lo que estamos haciéndole saber a la función bfs es saber cual es su principio, y cual es su final y cual es el grafo donde tiene que descubrir.

Esto hace que muchas veces en vez de buscar por el primer nodo pasado en la lista ya sabe por donde empezar a buscar, de tal manera que muchas veces las búsquedas se hacen mas rápidas por el propio hecho de saber donde buscar.

5.6)

El código funciona para la siguiente entrada tal que:

Lo primero que se hace es llamar a la función de shortest-path, a la cual le pasamos los argumentos de nodo de inicio, que es el nodo desde donde se empieza a buscar, nodo final, que es el nodo que queremos buscar, y la lista de listas que representan las conexiones entre los nodos.

Sabiendo esto, shortest-path llamará a bfs, pasándole el nodo final, a encontrar, el nodo donde tiene que empezar a buscar, siendo este list (list start), y todos los nodos conectados entre si.

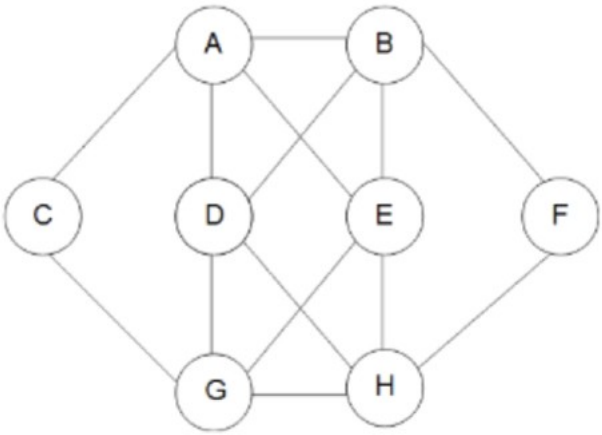
Como el argumento del nodo de inicio a partir de donde se empieza a buscar está metido ya en la queue de bfs, le indicamos cual es el primer nodo de la lista, y por tanto a partir de que nodo tiene que empezar a descubrir los demás nodos. Una vez que nuestra función bfs sabe esto, realiza exactamente el mismo proceso que hemos comentado en el apartado previo.

Obteniendo la siguiente salida:

```
[4]> (shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b f) (f)))
(A D F)
```

5.7) Para representar el siguiente grafo, tenemos que analizar las conexiones que tiene cada uno de los nodos que lo componen con sus adyacentes, el resultado fue l siguiente:

Para el grafo:



Creamos la siguiente llamada:

(shortest-path 'a 'f '((c g a)(a c d e b)(b a f e d)(f b h)(d a b h g)(e b h a g)(g c d h e)

(h g d e f)).

El resultado que hemos obtenido es inconcreto, creemos que es dado a que al haber ciclos en este grafo, la función entra en un bucle infinito en el que no para de añadir nodos a la cola para explorar de forma que nunca “termina” de analizar todos.

5.8) Implementamos las funciones que se nos piden, en este caso son una nueva versión de shortest-path llamada shortest-path-improved y bfs, llamada bfs-improved.