Parcial de Programación 3
4 de octubre de 2021

En recuadros con este formato aparecerán aclaraciones que cumplen una función explicativa pero que no eran requeridos como parte de la solución.

Ejercicio 1 (15 puntos)

Considere un programa cuyo código fuente se compone de una secuencia de instrucciones numeradas del 1 al n, las cuales son ejecutadas en orden. El código utiliza m variables, y para cada variable v_k se conoce el bloque B_k dentro del cual la variable es accedida (llamamos *bloque* a un intervalo de instrucciones consecutivas). Por ejemplo, si el bloque de una variable son las instrucciones $[i \dots j]$, únicamente las instrucciones comprendidas entre la i y la j inclusive acceden a ella.

Decimos que dos bloques B_r y B_k se solapan si $B_r \cap B_s \neq \emptyset$.

El compilador asigna a cada variable una celda fija de memoria donde se almacenará su contenido. Una celda puede ser asignada a más de una variable siempre que sus respectivos bloques no se solapen. Se desea realizar esta asignación de variables utilizando la menor cantidad de celdas de memoria posible.

(a) Dé un algoritmo ávido (*greedy*) que devuelva una asignación de las m variables a celdas de memoria utilizando la menor cantidad posible, respetando además las restricciones planteadas. Asuma que la cantidad de celdas disponibles es suficiente para que exista una asignación.

(b) Demuestre que su algoritmo es correcto. Repita cualquier argumento que utilice de los estudiados en el curso.

Solución:

(a) Sea d_i la cantidad de variables accedidas por una instrucción $i \in \{1, \dots, n\}$. Puesto que los bloques de dichas variables se solapan entre sí, todas éllas deberán ser asignadas a celdas de memoria diferentes, por lo que se requieren al menos d_i celdas diferentes. Por lo tanto, para asignar las m variables del programa, la cantidad mínima de celdas d requeridas necesariamente debe cumplir que $d \geq \text{máx}\{d_i\}_{1 \leq i \leq n}$. A continuación se presenta un algoritmo que realiza la asignación utilizando exactamente d celdas, es decir, la cantidad mínima posible, lo cual implica que la solución retornada por el algoritmo es óptima.

```
1  Algorithm AsignacionVariables
2      Ordenar las variables de manera creciente según el número de la primera instrucción de sus
         bloques, resolviendo empates arbitrariamente.
3      Sean v_1, ..., v_m las variables en el orden definido y B_1, ..., B_m sus respectivos bloques;
4      foreach j in 1...m do
5          Hacer C = {1, ..., d} igual al conjunto de celdas disponibles;
6          foreach i in 1...j-1 do
7              if B_i ∩ B_j ≠ ∅ then
8                  Eliminar del conjunto C la celda asignada a la variable v_i;
9              end
10         end
11         if C ≠ ∅ then
12             Tomar una celda cualquiera del conjunto C y asignarla a la variable v_j;
13         else
14             Dejar la variable v_j sin asignar;
15         end
16     end
17     Devolver la asignación calculada;
18 end
```

Figura 1: Algoritmo de asignación de variables a celdas de memoria.

(b) Probemos primero que ningúna variable queda sin ser asignada a una celda. Consideremos entonces una variable v_i cualquiera y su bloque de instrucciones B_i. Supongamos que luego de ordenar las variables, hay t variables que vienen antes que la variable v_i cuyos bloques se solapan todos con el bloque B_i. Puesto que los bloques de estas t variables más el de la variable v_i se solapan todos entre sí, esto implica que hay una instrucción que accede a cada una de estas $t+1$ variables. De lo dicho en la parte anterior se deduce entonces que $t+1 \leq d$, es decir, $t \leq d-1$, lo que implica que habrá al menos una celda libre de las d disponibles al momento de realizarse la asignación de la variable v_i, con lo cual quedará asignada a una celda.

Por último, probemos que si los bloques de dos variables v_i y v_j se solapan entre sí, entonces las variables son asignadas a celdas de memoria diferentes. En efecto, si denotamos a sus bloques como B_i y B_j respectivamente, y suponiendo sin perdida de generalidad que la variable v_i viene antes en el ordenamiento que la variable v_j, cuando el algoritmo considere el bloque B_j, la celda asignada a v_i será eliminada del conjunto de celdas disponibles ya que $B_i \cap B_j \neq \emptyset$, por lo que la variable v_j será asignada a una celda diferente a la de v_i.

Ejercicio 2 (20 puntos)

Una empresa de logística quiere actualizar su planta incorporando robots móviles que asistan en tareas de transporte. La planta puede verse como una grilla que cuenta con celdas libres por donde se puede transitar y celdas ocupadas donde no se puede. Esta información se tiene disponible como una matriz con n entradas que representan al estado de cada celda ($\{libre, ocupado\}$).

Las celdas son cuadradas de lado 1; el robot se posiciona en el centro de la celda de la grilla y se mueve de a una celda por vez pudiendo elegir hacerlo hacia cualquier celda vecina libre. Tenga en cuenta que al moverse entre una celda y su vecina en diagonal, el robot recorre una distancia de $\sqrt{2}$.

Asuma que existe un camino transitable entre cualquier par de celdas libres.

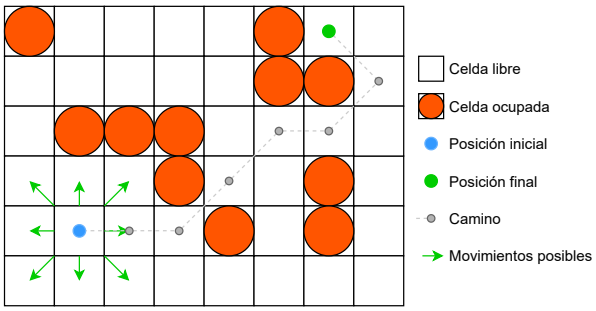


Figura: Grilla de ocupación del entorno de tamaño $n = 6 \times 8$.

(a) Escriba un algoritmo que permita que un robot encuentre uno de los caminos transitables más cortos desde su posición inicial hasta su destino.

Su algoritmo recibe una matriz como la mencionada anteriormente, las dimensiones de la matriz (filas, columnas), la posición inicial del robot (fila, columna) y la posición destino (fila, columna). Reescriba cualquier algoritmo que utilice de los presentados en el curso.

(b) Demuestre que su algoritmo admite una implementación con tiempo de ejecución que es $O(n \log n)$, siendo n la cantidad de entradas de la matriz. Enuncie los resultados teóricos del libro de referencia que utilice.

Solución:

El problema se modela con un grafo $G = (V, E)$ con un vértice por cada celda libre y una arista por cada par de celdas libres adyacentes. El costo de la arista es $\sqrt{2}$ o 1 según las celdas estén en diagonal o no. En el grafo se aplica el algoritmo de Dijkstra para encontrar el camino más corto desde el vértice que corresponde a la celda inicial hacia cada uno de los otros vértices. Para cada vértice se mantiene la distancia del camino conocido más corto (cualqueira de ellos si hay más de uno) desde el vértice de inicio, y el vértice inmediatamente anterior en ese camino, el 'padre'. Al terminar el algortimo de Dijkstra se construye el camino recorriendo la secuencia de padres desde el vértice fin hasta el inicio.

(a)

```
1  Algorithm CaminoMasCorto (M,filas,columnas,inicio,fin)
2      foreach celda libre en M do
3          agregar un vértrice en G
4      foreach v de V do
5          foreach w de V tal que v y w representan celdas adyacentes do
6              agregar w a la lista de adyacentes de v con costo l_vw √2 o 1
7      d[inicio] ← 0, d[v] ← ∞ para cada v ≠ inicio
8      S ← {inicio}
9      while S ≠ V do
10         (u, w) ← argmin_{(u',w')∈E,u'∈S,w'∈V\S} {d[u'] + l_u'w'}
11         S ← S ∪ {w}
12         d[w] ← d[u] + l_uv
13         padre[w] ← u
14     p ← fin
15     camino ← (p)
16     while p ≠ inicio do
17         p ← padre[p]
18         camino ← p . camino
19     Devolver camimo
20 end
```

Figura 2: Algoritmo para encontrar el camino de largo mínimo.

(b) Se mantiene las estructuras de listas de adyacencia, distancias y padres mediante arreglos bidimensionales por lo que se puede acceder en $O(1)$ a la información de cada vértice.

El ciclo de la línea 2 es $O(n)$ creando las listas de adyacentes vacías.

El ciclo de la línea 4 es $O(m)$. Cada vértice tiene a lo sumo 8 adyacentes, por lo que su grado es $O(1)$. Entonces el costo del ciclo es $O(n)$.

La línea 7 es $O(n)$.

El ciclo de la línea 9 es el algoritmo de Dijkstra visto en el libro de referencia, del que se conoce que es $O(m \log n)$, al que se le agregó la asignación del padre de w. Esta asignación tiene costo $O(1)$ por iteración y como hay $O(n)$ iteraciones el costo que se agregó es $O(n)$. Entonces el costo es $O(m \log n + n)$, que como $m = O(n)$ es $O(n \log n)$.

Finalmente para la construcción del camino se accede en cada iteración del ciclo de la línea 16 una vez a *padre* y se inserta en *Camino* lo cual tiene costo $\Theta(1)$. Este ciclo tiene $O(n)$ iteraciones por lo que su costo total también es $O(n)$.

El resto de los pasos tiene costo $O(1)$.

Entonces, como hay una cantidad de pasos independiente del tamaño de la entrada, el costo es el del paso de mayor costo, $O(n \log n)$.

Solución alternativa

(a) Se crea y mantiene una cola de prioridad, Candidatos. Sus elementos son triplas (v, d, p), donde v es una celda, d es la distancia del camino más corto conocido desde la posición inicial hasta v, y p es la celda inmediatamente anterior a v en ese camino. Si p está indefinido se representa con $\perp$. Las operaciones de la cola de prioridad son:

- **Crear** que la crea sin elementos,
- **Agregar** que inserta una tripla,
- **Minimo** que devuelve una tripla cuya distancia tenga valor mínimo,

- **EliminarMinimo** que remueve la tripla devuelta por **Minimo**,
- **Actualizar** que, dada una tripla (v, d', p'), si v está en la cola de prioridad en una tripla (v, d, p) y se cumple $d' < d$, entonces (v, d', p') sustituye a (v, d, p).

Se crea y mantiene un conjunto S con las celdas para los que se encontró el camino más corto desde la posición inicial. Sus elementos son pares (v, p), donde, tal como en la cola de prioridad, p es la celda inmediatamente anterior a v en ese camino. Esta celda p también puede considerarse el *padre* de v en un árbol obtenido por el algoritmo. El conjunto tiene operaciones para crear, incluir, determinar si una celda está en él (o sea, si la celda es el primer componente de algún par del conjunto), y obtener el padre de una celda v que está en el conjunto (o sea, el segundo componente del par (v, p)).

El resultado que se devuelve es **Camino**, que es una lista de celdas. Se inserta al inicio de la lista.

```
1  Algorithm CaminoMasCorto (M,filas,columnas,inicio,fin)
2      Crear S
3      Crear Candidatos
4      foreach v tal que M[v] = libre, v ≠ inicio do
5          Agregar en Candidatos (v, ∞, ⊥)
6      (v, d, p) ← (inicio, 0, ⊥)
7      while v ≠ fin do
8          Incluir (v, p) en S
9          foreach w adyacente a v, M[w] = libre y no está en S do
10             Actualizar (w, d + c, v) en Candidatos, donde c es la distancia entre v y w
11         (v, d, p) ← Minimo (Candidatos), EliminarMinimo (Candidatos)
12     Crear Camino
13     p ← fin
14     Insertar p en Camino
15     while p ≠ inicio do
16         p ← padre de p en S
17         Insertar p en Camino
18     return Camino
19 end
```

Figura 3: Algoritmo para encontrar el camino de largo mínimio.

(b) La cantidad de celdas libres es $O(n)$.

El conjunto S se puede representar mediante una matriz de las mismas dimensiones de M, en cuyas entradas se mantiene el padre del vértice correspondiente, con valor incial $\perp$. De esta forma las operaciones en S son $\Theta(1)$. La creación de S es entonces $O(n)$.

La cola de prioridad se implementa con un heap binario. El ciclo de la línea 4 es $O(n)$, porque simplemente se asigna cada elemento del heap en $O(1)$.

En cada iteración del ciclo de la línea 7 se incluye un elemento en S y se lo remueve de `Candidatos`, donde no se vuelve a insertar. Por lo tanto la canitdad de iteraciones es $O(n)$. En cada iteración se incluye un elemento en S en $O(1)$, se obtiene el mínimo de un heap en $O(1)$, se elimina el mínimo de un heap en $O(\log n)$, y se ejecuta el ciclo de la línea 9. En este ciclo se consulta si w está en S en $O(1)$ y se hacen actualizaciones en un heap, cada una de las cuales es $O(\log n)$. Pero la cantidad de actualizaciones no es mayor a 8, $O(1)$, por lo que el costo total de todas las actualizaciones es $O(\log n)$, y del ciclo interno es $O(\log n)$. Entonces todo el ciclo de la línea 7 es $O(n \log n)$.

Finalmente para la construcción del camino se accede en cada iteración del ciclo de la línea 15 una vez a S y se inserta en *Camino* lo cual tiene costo $\Theta(1)$. Este ciclo tiene $O(n)$ iteraciones por lo que su costo total también es $O(n)$.

Entonces, como hay una cantidad de pasos independiente del tamaño de la entrada, el costo es el del paso de mayor costo, $O(n \log n)$.

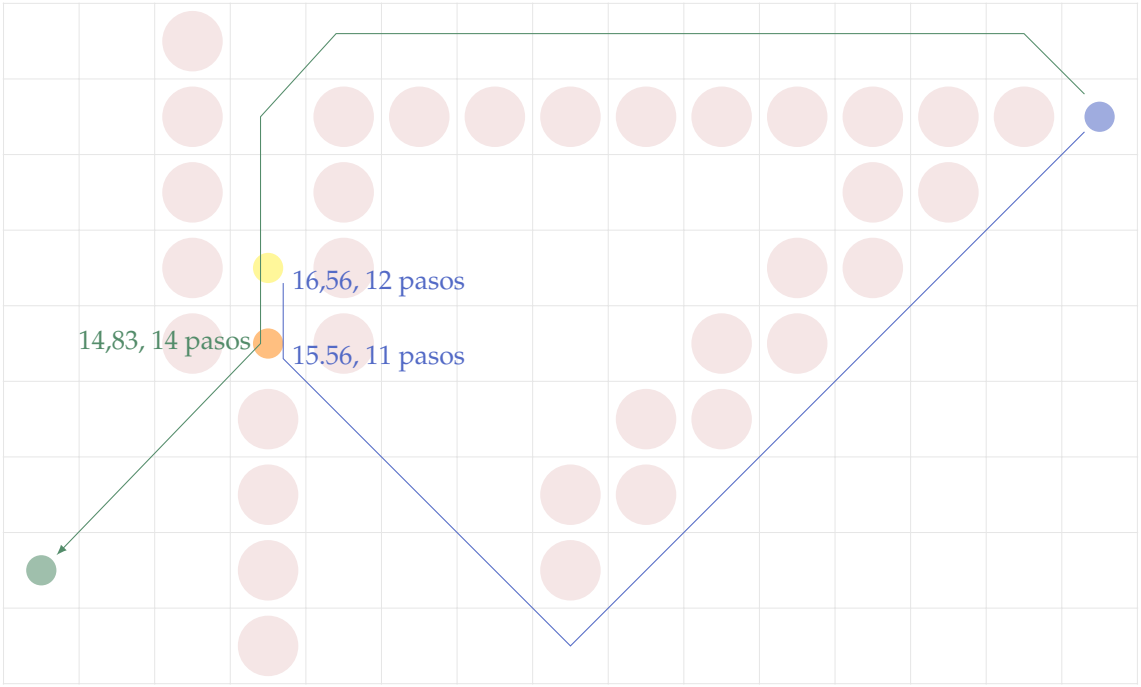
Ejemplo de estrategias alternativas que no resuelven el problema.

En el ejemplo se ve que la solución debe usar el camino indicado con la línea verde.

Se debe notar que en el primer paso, a pesar de poder acercarse al objetivo según ambas coordenadas, la solución se aleja según la coordenada vertical, por lo que se debe descartar una posible estrategia de intentar acercarse todo lo que sea posible en cada paso.

La solución requiere 14 pasos para llegar a la celda marcada con naranja, a pesar de que hay un camino, el indicado con azul, con 11. Esto muestra que no son aplicables estrategias basadas en `BFS`.

Tampoco sirve una modificación de `BFS` que actualice si se llega a una celda ya visitada por un camino más corto que aquel por el cual se había llegado antes. Se debe notar que el camino indicado en azul también requiere menos pasos para llegar a la celda que está marcada con amarillo. En la alternativa aquí sugerida al seguir el camino verde se podría modificar la distancia y el camino para llegar a la celda amarilla cuando se procesa la celda inmediatamente superior, pero la recorrida termina, por lo que no se llega a modificar la celda naranja. Si la modificación del algoritmo incluyera volver a encolar el vértice correspondiente la celda modificada, entonces dejaría de cumplirse los análisis de tiempo de ejecución porque se volverían a recorrer aristas. En este ejemplo eso pasaría al menos con todos lo que están a la izquierda de la celda naranja.



Ejercicio 3 (15 puntos)

Sea A' una secuencia de tres o más naturales ordenada de forma ascendente, tal que entre cada par de elementos consecutivos la diferencia es constante.

Ahora considere A, la secuencia resultante de eliminar un elemento de A' que no sea ni el primero ni el último. A tiene largo n, con n potencia de 2.

Ejemplo: $A' = [7, 11, 15, 19, 23, 27, 31, 35, 39]$, la diferencia entre cada elemento es 4, y $A = [7, 11, 15, 19, 23, 31, 35, 39]$.

(a) Dé un algoritmo de tipo **Divide y Vencerás** que, dada una secuencia A como la definida anteriormente, devuelva el elemento faltante. Su algoritmo debe admitir una implementación con tiempo de ejecución $O(\log n)$.

(b) Muestre que su algoritmo admite una implementación con tiempo de ejecución que es $O(\log n)$. Puede utilizar resultados presentados en el libro de referencia.

Sugerencia: represente las instancias del problema como un par de indices de inicio y fin dentro de A.

Solución:

(a) Para resolver el algoritmo primero tenemos que conseguir la diferencia. Para eso una opción es comparar la diferencia entre el primer y el ultimo elemento. Es decir la diferencia es $d = (A[n] - A[1])/n$
Luego dado que tenemos la diferencia d.
Dado un problema de tamaño **n**, lo dividimos en dos subproblemas y eso nos da tres categorías:

- Encontrar el elemento en el subarreglo $A[1..n/2]$
- Encontrar el elemento en el subarreglo $A[n/2 + 1..n]$
- Si el elemento no esta en ningun subarreglo, necesariamente tiene que estar en la combinación de los dos, por lo que comparamos la diferencia entre $A[n/2]$ y $A[n/2 + 1]$. Por lo que el elemento faltante es $A[n/2] + d$

Para saber si un elemento esta faltante en un rango del arreglo entre el elemento i, j. Con $i < j$ comparamos la diferencia entre los elementos de los extremos del arreglo.
$(A[j] - A[i])/(j - i)$. Si la diferencia es igual a d entonces el elemento faltante no se encuentra en ese arreglo, si es mayor el elemento faltante se encuentra ahi.
Si en un subarreglo se encuentra que falta un elemento, se realiza la misma técnica en ese subarreglo, es decir, se aplica nuevamente la recursion.

```
1  Algorithm ElementoPerdido
       Data: palabra
       Result: largo
2      if n == 2 then
3          return A[1] + d
4      else
5          diferenciaIzq = A[n/2] - A[1]
6          diferenciaDer = A[n] - A[n/2+1]
7          if diferenciaIzq >diferenciaDer then
8              return ElementoPerdido A[1..n/2]
9          if diferenciaIzq <diferenciaDer then
10             return ElementoPerdido A[n/2+1..n]
11         return A[n/2] + d
12     end
13 end
```

Figura 4: Algoritmo para encontrar el elemento faltante, dado que se tiene la diferencia d.

(b) Para demostrar que el algoritmo admite una implementación $O(logn)$, primero necesitamos una relación de recurrencia. Sea $T(n)$ la función que denota el tiempo de ejecución para una entrada de tamaño n para el peor caso.

Obtener la distancia es orden constante y se realiza por fuera del algoritmo que vamos a analizar a continuación. Esto no afecta la complejidad del algoritmo global.
Para analizar la relación de recurrencia, vemos que el paso base de la misma es constante, por lo que esta acotado por c, entonces $T(2) \leq c$, $c > 0$.

A su vez, en cada llamada del algoritmo, hay tres casos posibles, el elemento esta en una mitad, el elemento esta en la otra mitad o el elemento se encuentra entre los dos subarrelgos. Analizaremos los primeros dos, ya que son los casos donde se da el peor caso. Como dijimos, en ambos se ejecuta el algoritmo, pero con el tamaño de la entrada siendo la mitad más un tiempo constante c. $T(n) \leq T(n/2) + c$.

Para que efectivamente el algoritmo sea $O(logn)$, cada llamada recursiva es siempre sobre el mismo arreglo inicial con los indices adaptados para la nueva llamada (no se puede realizar una copia del subarreglo en cada llamada recursiva, esa operación provocaría que el tiempo de ejecución aumente).

Haremos un razonamiento inductivo para demostrar que es, efectivamente de orden $O(logn)$.
El paso base del razonamiento inductivo seria:

$T(2) \leq c$, por lo que se cumple que es O(log n)

Para el paso inductivo asumiremos que se cumple para todo $m < n$, entonces por definición $T(n/2) \leq c.log(n/2)$. Sustituyendo $T(n/2)$ tenemos que:
$T(n) \leq c.log(n/2) + c = c.log(n) - c.log(2) + c = c.log(n) - c + c = c.log(n)$.
Por lo que $T(n)$ es $O(logn)$.