Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 1

Integrante	LU	Correo electrónico
Laura Muiño	399/11	mmuino@dc.uba.ar
Martín Santos	413/11	martin.n.santos@gmail.com
Luis Toffoletti	827/11	luis.toffoletti@gmail.com
Florencia Zanollo	934/11	florenciazanollo@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Ejercicio 1

1.1. Descripción

El sistema utilizado por Pascual consiste en cargar paquetes en camiones, de acuerdo al orden de llegada. Cada paquete se intenta cargar en el camión con menor peso (con peso mayor que cero), si no es posible se guarda en un nuevo camión.

El problema pide que se implemente el sistema utilizado por Pascual, con L (el peso máximo soportado por todos los camiones), n (la cantidad de paquetes a cargar) y p_1, p_2, ..., p_n (pesos de los paquetes) los parámetros de entrada. Se supone que los paquetes no superan la capacidad máxima de los camiones. Hagamos un ejemplo a modo de ilustración:

Sea L = 25, n = 7 y $p_1 = 25$, $p_2 = 13$, $p_3 = 18$, $p_4 = 8$, $p_5 = 12$, $p_6 = 4$ y $p_7 = 1$. De acuerdo al método del buen hombre, la solución es utilizar cuatro camiones con 25, 21, 18 y 17 sus pesos respectivos. El primer camión contiene el primer paquete, el segundo camión contiene al segundo y al cuarto paquete, el tercer camión contiene el tercer paquete, el cuarto camión contiene al quinto, sexto y séptimo paquete.

Para desarrollar un algoritmo que llegue a la misma solución que obtendría el sistema de Pascual, lo que hicimos fue seguir los mismos pasos que realiza dicho sistema. Para eso fue necesario reducir el enunciado a ciertos puntos básicos, que nos llevarían a la creación de nuestro modelo.

Las características del problema se reducen a lo siguiente:

1 - Los paquetes se encolan según el orden de llegada, por lo tanto el primer paquete que llega es el primero que se guarda en un camión.

2 - Siempre hay camiones disponibles, y todos ellos poseen el mismo límite de carga.

3 - Ningún paquete tiene un peso mayor al límite de carga.

4 - Se desea además conocer el número de camiones utilizados.

5 - El método de Pascual: Cada paquete intenta colocarse dentro de un camión que ya esté con al menos una carga, Pascual intenta cargarlo en aquel que lleve menor peso, de no ser posible utiliza un nuevo camión.

1.2. Pseudocódigo

Con el problema caracterizado buscamos modelarlo, veamos punto por punto:

1 - Como importa el orden, necesitamos una lista para representar los paquetes,y como lo que nos interesa es saber el peso de cada uno, utilizamos una lista de enteros.

2 - El peso límite estará reflejado por un entero y como es el mismo para todos, lo definimos de forma global.

3 - Este dato junto al punto 2, nos determina que todos los paquetes serán cargados y que siempre que se evalúe un paquete será ubicado. Es importante porque nos asegura que siempre recorremos la lista de paquetes una única vez, y en el número de paquetes.

4 - Cada vez que se agrega un nuevo camión tenemos que asegurarnos de contabilizarlo, para devolverlo en la salida.

5 - De este punto se desprenden muchas decisiones, algunas se justifican por la necesidad de alcanzar una cierta cota de complejidad, eso se explica más adelante:

   a - Nos importa el peso que carga cada camión de los existentes, definimos una lista de enteros que los representen, se irán agregando según el orden de creación.

   b - Al momento de evaluar donde poner un paquete, necesitamos saber el peso del que menor carga tiene (de los que posean carga), para ésto podríamos recorrer la lista de camiones y buscar el menor, pero para reducir la complejidad empleamos un heap con tuplas de dos enteros, una para el peso y otra para el índice del camión con respecto a la lista detallada en el punto a. El primero del heap será el de menor peso.

   c - Nos queda simplemente recorrer la lista de paquetes en orden, con la información que nos da el primer elemento del heap sabemos si podemos colocarlo en el camión con menor carga, sino se coloca en uno nuevo y se contabiliza en el número de camiones. Hay que tener en cuenta que al cargar un paquete se deben actualizar el heap y la lista de camiones, de forma que la información sea consistente.

La salida es la variable que cuenta camiones seguida de la lista del punto 5-a que contiene los pesos de los camiones según el orden de carga de su primer paquete.

A continuación se muestra el pseudocódigo del algoritmo principal y se muestran los puntos descriptos anteriormente.

Algorithm 1 Pascual

```
1:  procedure PASCUAL(L, n, Paquetes)        ▷ L= peso límite, n= cant paquetes, Paquetes= peso de c/paquete
2:      Camiones ← CrearVectorVacio()
3:      HeapCamiones ← CrearHeapVacio()      ▷ HeapCamiones los ordena por peso, quedando el mínimo primero
4:      k ← 0                                                              ▷ k= cantidad de camiones
5:      for all p in Paquetes do
6:          if PesoMinimo(HeapCamiones) + p ≤ L then
7:              PesoMinimo(HeapCamiones) ← PesoMinimo(HeapCamiones) + p
8:              Camiones[IndicePesoMinimo(HeapCamiones)] ← PesoMinimo(HeapCamiones)
9:                                                  ▷ actualizo el peso del camión en el vector
10:         else
11:             k ← k + 1
12:             AgregarNuevoCamion(Camiones, p)                  ▷ agrego un nuevo camion con peso=p
13:             AgregarNuevoCamion(HeapCamiones, p, k)   ▷ agrego un nuevo camion con peso=p e índice=k
14:         end if
15:     end for
16:     return k, Camiones
17: end procedure
```

1.3. Demostración

Por lo enunciado anteriormente podemos afirmar que nuestro algoritmo sigue los mismos pasos que el sistema de Pascual, es decir lo modela correctamente, y por lo tanto la manipulación de los datos de entrada deriva en un resultado correcto.

1.4. Complejidad

Para programar este algoritmo utilizamos un par de contenedores de datos definidos en la Standard Template Library: vector y priority_queue.
Las complejidades de las funciones de dichos contenedores fueron sacadas de sus respectivas documentaciones.

Para ver la documentación del contenedor vector: http://www.cplusplus.com/reference/vector/vector/

Para ver la documentación del contenedor priority_queue: http://www.cplusplus.com/reference/queue/priority_queue/

Análisis de complejidad en el peor caso:

Inicio del algoritmo

Líneas 2 y 3: CrearVectorVacio()[1] y CrearHeapVacio()[2] tienen complejidad constante O(1).

Línea 4: Asignar un valor a una variable también O(1).

Línea 5 **Inicio del For**: Por cómo está definido el ciclo entra una vez por cada paquete. Complejidad lineal O(n), n=cantidad de paquetes.

[1] http://www.cplusplus.com/reference/vector/vector/vector/
[2] http://www.cplusplus.com/reference/queue/priority_queue/priority_queue/

Línea 6 **Guarda del If**: Obtener el mínimo del heap es O(1).[3] Luego la suma y la comparación de enteros son constantes. Complejidad total de la guarda: O(1) + O(1) = O(1)

Líneas 7 y 8 **Cuerpo del If**: Para poder modificar el elemento del heap debemos obtenerlo y luego reincorporarlo. Es decir, llamamos una vez a la función 'pop' de priority_queue (esta función es O(2.log(k)), k= cantidad de camiones hasta el momento) y una vez a la función 'push' de priority_queue (esta función es O(log(k)))
Luego debemos actualizar el peso del camión en el vector. Acceder al elemento y modificarlo es O(1).
Complejidad total del cuerpo del If: O(2.log(k)) + O(log(k)) + O(1) = O(3.log(k)) = O(log(k))

Líneas 11, 12 y 13 **Cuerpo del Else**: Modificar el valor de una variable es O(1).
Luego creamos un nuevo camión O(1), lo agregamos al vector de camiones O(1) y al heap (función 'push' de priority_queue) O(log(k)).
Complejidad total del cuerpo del Else: O(1) + O(1) + O(log(k)) = O(log(k))

Línea 14 **Fín If**: Complejidad en el peor caso del If = O(log(k))

Línea 15 **Fín For**: Complejidad total: O(n).O(log(k)) = O(n.log(k))

Complejidad total del algoritmo: O(1) + O(1) + O(1) + O(n.log(k)) = **O(n.log(k)) Siendo n=cantidad de paquetes y k=cantidad de camiones.**
En el peor caso hay un camión por cada paquete, entonces a lo sumo k=n, si se quiere expresar la complejidad respecto sólo a los datos de entrada, una cota un poco exagerada (ya que el heap se va creando y sólo al final contiene los n camiones a la vez) sería **O(n.log(n))** Siendo n=cantidad de paquetes

[3] http://www.cplusplus.com/reference/queue/priority_queue/top/

2. Ejercicio 2

2.1. Descripción

2.2. Pseudocodigo y análisis de complejidad

El algoritmo que resuelve el problema esta representado por el siguiente pseudocodigo:

Algorithm 2 Resolver

```
1:  procedure RESOLVER(n, cursos)▷ n= cantidad de cursos, cursos= vector de tuplas con inicio y fin de cada curso
2:      sort(cursos)                                  ▷ De menor a mayor de acuerdo a las segundas coordenadas
3:      final ← 0
4:      for i ← 0, i < cursos.size(), ++i do
5:          if cursos[i].first.first > final then
6:              cursosRes.pushBack(cursos[i])
7:              final ← cursos[i].first.second
8:              return cursosRes
9:          end if
10:     end for
11: end procedure
```

Pasemos a analizar la complejidad:

- En la linea 2, usamos el algorimo sort definido en la biblioteca estandar de c++ (http://www.cplusplus.com/reference/algorithm/sort/?kw=sort). Su complejidad es N*log2(N), donde n es la cantidad de elementos del vector de entrada.
- En la linea 4, el for realiza N iteraciones.
- Luego la complejidad total es $N * log2(N)$, estrictamente menor a N^2.

2.3. Demostración

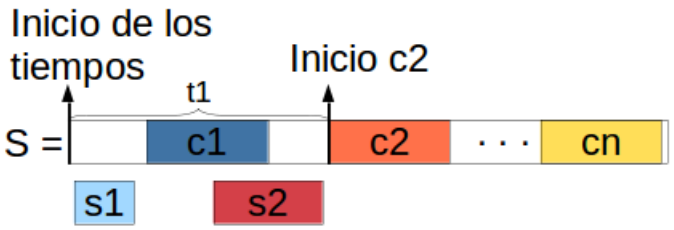
Sea S una solución dada por nuestro algoritmo, supongo que existe S* tal que tiene al menos un curso más que S. Es lo mismo que decir que podemos sacar k cursos de S y agregar, como mínimo, k+1 (con 0<k≤n, n=#S).

Si se quiere sacar un curso y agregar dos:
Supongo que podemos sacar el primer curso (c1) de S y poner dos en su lugar. Para que dos cursos (s1, s2) quepan en el tiempo que quedó libre (t1) tienen que cumplir las siguientes características:

1. No se deben solapar entre sí (sino no podrían agregarse juntos).
2. Sus inicios y fines deben estar dentro del rango de tiempo libre (t1).

Con esto lo podemos dividir en casos:

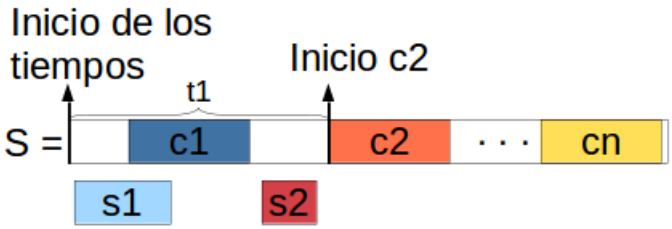
Caso A: s1 no se superpone con c1



Para que esto pase s1 debe terminar antes de c1, como nuestro algoritmo los coloca según el orden de sus finales esto

es absurdo; dado que el algoritmo hubiera colocado primero a s1.

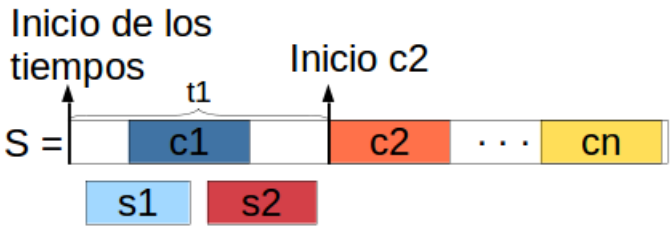
Caso B: s2 no se superpone con c1



Para que esto pase s2 debe empezar luego del final de c1, y como deben estar dentro del rango, debe terminar antes de c2. Esto es absurdo porque en este caso, al no haber conflicto ni con c1 ni con c2, el algoritmo lo hubiera colocado entremedio de estos.
Observación: en este caso, el algortimo hubiese elegido s1 en vez de c1 pero esto no modifica la cantidad de cursos de S.

El caso en que ambos (s1 y s2) no se superponen con c1 es trivial luego de las explicaciones de los casos A y B.
Observación: estos casos donde no se superponen prueban que no se puede "sacar"0 cursos y agregar 1.

Caso C: ambos (s1 y s2) se superponen con c1



De estas características se desprende que la única forma de que estén dispuestos s1 y s2 es que s1 finalice dentro del rango de c1 y s2 comience dentro del rango de c1.
Por lo tanto s1 terminaría antes que c1, es decir, nuestro algoritmo lo hubiese colocado en lugar de c1 y sería el primer curso de la solución. Entonces S no sería la solución dada por el algoritmo.

Por lo tanto c1 es una elección correcta y no hay forma de cambiar el curso por otros dos.

Si se quiere sacar 2 cursos y agregar 3:
Por la explicacion anterios sabemos que c1 está correctamente elegido, entonces la única forma de colocar 3 cursos donde antes había dos es reemplazar c2 por dos cursos. Probandolo de la misma forma que para c1 se llega a un absurdo.

Por lo tanto c1 y c2 son elecciones correctas y no hay forma de cambiar dos cursos por otros tres.

Si se quiere sacar k cursos y agregar k+1:
Extendiendolo de esa forma se llega a que no hay una manera de colocar k+1 cursos.

Nota: Esto se generaliza para el caso de tomar cursos aleatorios (no sólo adyacentes) ya que la explicación es esencialmente la misma. Elegimos hacerlo en orden sólo por una cuestión de claridad.

3. Ejercicio 3

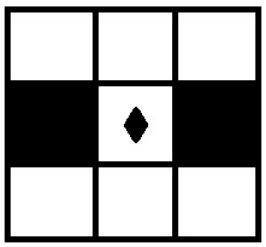
3.1. Descripción

Dada una grilla de mxn casillas que representan el piso de un museo, se quiere colocar un sistema de detección basado en sensores de manera tal que cada una de las casillas quede cubierta por algún sensor. Que quede cubierto significa que haya un sensor sobre el casillero o que sea apuntado por al menos uno. Cada sensor tiene un costo y la idea es desarrollar un algoritmo que encuentre una solución que lo minimice cumpliendo con lo pedido. Dentro del museo disponemos de tres tipos de casilleros:

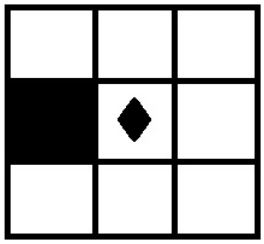
- Casilleros simples: debe haber al menos un laser que los cubra.
- Casilleros importantes: requiere ser apuntado por dos lasers.
- Paredes: Su única función es bloquear las señales emitidas por un sensor.

El problema pide además que no haya un sensor apuntando a otro ni tampoco haber dos sensores en el mismo casillero.

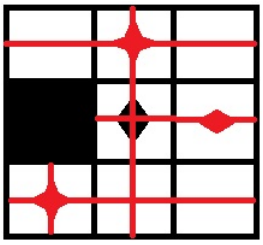
Dada cierta configuración inicial podría suceder el caso de que no exista una solución posible. Por ejemplo:



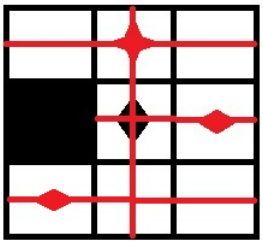
Es imposible colocar sensores de forma tal que el casillero importante sea apuntado por dos sensores ya que de ninguna manera se puede lograr que sea apuntado de forma horizontal por la presencia de las paredes. Removiendo alguna de las paredes obtenemos una grilla en la cual se puede obtener una solución:



Una solución posible para esta configuración es la siguiente:



Sin embargo esta solución no es óptima ya que el costo total es de $16000 (dos sensores cuatridireccionales y uno bidireccional) y es posible encontrar una mejor:



Aquí el costo es de $14000 (dos sensores bidireccionales y uno cuatridireccional).

3.2. Explicación del Algoritmo

La idea del algoritmo es realizar un backtrack sobre los casilleros simples del museo ya que son los únicos casilleros en donde podemos colocar sensores. Para cada casillero tenemos cuatro opciones:

1 Colocar un sensor bidireccional horizontal

2 Colocar un sensor bidireccional vertical

3 Colocar un sensor cuatridireccional

4 No colocar nada

El algoritmo prueba los cuatro casos para cada casillero y marca los casilleros restantes dependiendo de la elección tomada. En caso de elegir las opciones 1,2 o 3 se marcan todos los casilleros cubiertos por el sensor como usados. Si, por el contrario, decide no colocarse nada solo se marca el casillero actual y se vuelve a llamar a la función Backtrack.

El algoritmo dispone de las siguientes podas:

- Costo: el algoritmo verifica que la solución parcial construida no supere el costo de la mejor solución hallada hasta el momento. En caso de no haber encontrado ninguna solución hasta ese instante la poda no tiene efecto.
- puedoColocarSensor: verifica si un sensor puede colocarse en un casillero dado, es decir, chequea que no apunte a alguno de los sensores colocados.
- cumpleHastaElMomento: chequea para todos los casilleros importantes si en la solución parcial construida hasta el momento es posible hacer que el casillero sea apuntado por dos sensores. En caso de que no sea posible se corta con la rama de decisión.

Para modelar el problema utilizamos el siguiente struct:

```
struct Problema {
vector<vector<int>> _matriz
vector<Casillero> _casilleros
vector<Casillero> _casillerosImportantes
vector<vector<int>> _matrizRes
int _costo
}
```

en donde:

- _matriz representa el piseo del museo y es en donde se carga la configuración inicial.
- _casilleros es un vector de Casillero (simples) y Casillero es un $pair < int, int >$ que indica las coordenadas en la _matriz.
- _casillerosImportantes es un vector de casilleros importantes.
- cuando se encuentra una solución, _matrizRes almacena la grilla que representa el piso del museo con los sensores colocados y además se actualiza el valor de _costo.

Para resolver el problema realizamos el siguiente algoritmo:

A continuación presentamos un pseudocódigo de la función Backtrack:

Algorithm 3 Resolver

```
1: procedure RESOLVER
2:     cargarCasilleros()                ▷ carga el vector de _casilleros y _casillerosImportantes teniendo en cuenta la configuración inicial.
3:     casillerosUsados ← CrearVectorDelTamaoDe_Casilleros() ▷ Inicializa el vector casillerosUsados en 0 para todas las posiciones.
4:     int costo = 0                     ▷ Variable cuya función es almacenar el costo de una solución parcial
5:     _costo = −1                       ▷ Es inicializado en -1 para marcar que no hay solución válida hasta el momento
6:     Backtrack(casillerosUsados, costo)
7: end procedure
```

Algorithm 4 Backtrack

```
1: procedure BACKTRACK(vector < int > casillerosUsados, intcosto)   ▷ casillerosUsados: casilleros que fueron cubiertos hasta el momento, costo: dinero acumulado hasta el momento.
2:     vector < bool > casillerosUsadosViejo   ▷ vector auxiliar para que sea posible volver a una rama anterior dentro de nuestro arbol de decisiones
3:     int aux
4:     if ¬hayMas(casillerosUsados) then
5:         if esSolucion() ∧ costo < _costo then
6:             _matrizRes = _matriz                                   ▷ guardo el resultado
7:             _costo = costo                                         ▷ actualizo el costo óptimo hasta el momento
8:         end if
9:     else
10:        if costo < _costo ∧ cumpleHastaElMomento(casillerosUsados) then
11:            casillerosUsadosViejo = casillerosUsados               ▷ Guardo una copia antes de que sea modificado.
12:            for all c ∈ _casilleros do
13:                if ¬usado(c, casillerosUsados) then
14:                    for all s ∈ Sensores do
15:                        if puedoColocarSensor(c, s, _matriz) then
16:                            InsertarSensor(c, s_matriz)
17:                            marcarCasilleros(c, s, casillerosUsados)
18:                            if esSensorBidireccional(s) then
19:                                backtrack(casillerosUsados, costo + 4000)
20:                            else
21:                                backtrack(casillerosUsados, costo + 6000)
22:                            end if
23:                            casillerosUsados = casillerosUsadosViejo
24:                        end if
25:                    end for                                        ▷ Caso en el que no se inserta ningún sensor.
26:                    marcarCasillero(c, casillerosUsados)
27:                    backtrack(casillerosUsados, costo)
28:                    casillerosUsados = casillerosUsadosViejo
29:                end if
30:            end for
31:        end if
32:    end if
33: end procedure
```

en donde:

- hayMas(casillerosUsados) verifica si hay algun casillero sin ser cubierto.
- esSolucion() chequea si la disposición actual de los sensores es efectivamente un resultado válido. Se asegura que no haya sensores apuntándose entre si, que todos los casilleros se encuentren cubiertos y que los casilleros importantes sean apuntados por dos sensores.
- usado(c,CasillerosUsados) verifica que el casillero c está cubierto.

- puedoColocarSensor(c,s,_matriz) utilizando _matriz verifica si es posible agregar el sensor s en el casillero c sin apuntar a otro sensor.
- InsertarSensor(c,s,_matriz) inserta el sensor s en el casillero c de _matriz.
- marcarCasilleros(c,s,casillerosUsados) marca como cubiertos a todos los casilleros apuntados por s.

3.3. Correctitud del algoritmo

El algoritmo es correcto ya que para cada uno de los casilleros disponibles prueba colocar los tres sensores presentes y no colocar nada. A su vez las podas no restringen posibles soluciones porque se encargan simplemente de hacer o verificar que la rama de decisiones tomadas sea válida. Si se analizan todas las soluciones es lógico pensar que el algoritmo va a encontrar una óptima.

3.4. Complejidad

Para analizar la complejidad de este algoritmo es necesario evaluar el peor caso posible. Para cada casillero disponemos de 4 opciones: colocar 3 sensores o no colocar ninguno. En algún momento el algoritmo tomará la rama de decisión de no colocar absolutamente nada en ningún casillero ya que podría ocurrir que no dispongamos de casilleros importantes por lo que la poda cumpleHastaElMomento no tendría efecto. Lo mismo sucede para la poda de costo porque el costo es nulo en todo momento. Para llegar a esta rama de decisión es necesario que el algoritmo probara colocando los 3 sensores en cada casillero previamente por lo cual tendremos 4 opciones para cada casillero. En el peor caso posible tendremos m*n casilleros en donde colocar sensores (suponiendo que no hay paredes) por lo cual la complejidad será $\mathcal{O}(4^{m*n})$