Universidad de Buenos Aires
Facultad De Ingeniería
Año 2019 – 2.do Cuatrimestre

(75.29 / 95.06) Teoría de Algoritmos

TRABAJO PRÁCTICO N.º 1
Grupo: "Los Fieles de Cardozo"
11 de Septiembre de 2019

Integrante	Padrón	Correo electrónico
Salvador, Yago	# 99 725	yagosalvador@gmail.com
Nenadovit, Emmanuel Angel	# 91 734	emmanuelnenadovit@gmail.com
Untrojb, Kevin	# 97 866	oliverk12@hotmail.com
Bocchio, Guido	# 101 819	guido@bocch.io

1. Resumen

En el presente informe se expone el análisis y resultados a cada parte individual de la consigna. La primera parte consiste en analizar distintos tipos de solución a un mismo problema y el analisis de estas soluciones. La segunda parte consiste en el diseño de una estrategia para minimizar el tiempo requerido en un problema. En la tercera parte se analiza el uso del algoritmo desarrollado por David Gale y Lyorn Shapley, para un problema conocido como *"Stable Matching Problem"*, en el marco de otro problema inspirado por el programa *New England Program for Kidney Exchange*.

2. Múltiples soluciones para un mismo problema

El enunciado dice que se tiene un vector de N posiciones. En el mismo se encuentras los primeros M números naturales ordenados en forma creciente. M es mayor igual a N. En el vector no hay números repetidos. Se desea obtener el menor número no incluido.

2.1. Primera Solución

La primera solución consiste en comenzar desde el primer elemento del vector e incrementando de a una posición, verificar hasta encontrar el primer salto en la numeración o hasta recorrer todos las posiciones del vector. Si no hay faltantes retorna $n + 1$

Algoritmo 1 Primera Solución

```
array[1:n]
para i in 1:n hacer
    si array[i] ≠ i entonces devolver i
devolver n+1
```

Este algoritmo realiza una comparación por cada uno de los elementos. Como la complejidad algorítmica de cada comparación es a $\mathcal{O}(1)$, y se incurre en una a lo sumo una vez por cada elemento, se obtiene que este algoritmo es $\mathcal{O}(n)$.

2.2. Segunda Solución

La segunda solución consiste en comenzar desde el 1 hasta el número natural m, realizar una búsqueda binaria para determinar si se encuentra o no el elemento. Si no se encuentra ese es el primer número faltante.

Algoritmo 2 Segunda Solución

```
array[1:n]
para i in 1:n hacer
    si IsInTheArray(array,n,0,i) entonces devolver i
función IsInTheArray(array, high, low, i)
    si high < low entonces devolver False
    mid = (high + low)/2
    si array[mid] < i entonces
        devolver IsInTheArray(array,hight,mid+1,i)
    si array[mid] > i entonces
        devolver IsInTheArray(array,mid−1,low,i)
    si no
        devolver True
```

En esta solución el algoritmo a lo sumo invoca m veces a una función de búsqueda binaria. Como la complejidad algorítmica de un búsqueda binaria es de orden logarítmico en el tamaño del vector, en este caso resulta $\mathcal{O}(log(n))$. Por consiguiente, la segunda solución pertenece al orden de complejidad $\mathcal{O}(m \log(n))$

2.3. Tercera Solución

La tercera solución se basa en buscar el elemento del medio del vector. Si ese elemento es igual al valor de su posición en el vector (comenzando a contar desde 1) significa que no hay números faltantes en la primera mitad. Se repite este procedimiento descartando esta mitad. Si, por el contrario, el elemento es mayor al de su posición en el índice, indica que hay un faltante en esta primera mitad y se procede a descartar la segunda. Este procedimiento se repite $\mathcal{O}(log(n))$ veces. El número faltante será el correspondiente a esa posición en el vector original. En el caso de haber finalizado en la última posición del vector y su valor coincide con la posición, el número faltante será $n + 1$.

Algoritmo 3 Tercera Solución

```
array[1:n]
num =GetMissingNumber(array,n,0)
si num ≠ 0 entonces devolver num
devolver n + 1
función GetMissingNumber(array, high, low)
    si high = low entonces
        si array[high]=high entonces devolver high
        devolver 0
    mid = (high + low)/2
    si array[mid]=mid entonces devolver GetMissingNumber(array,high,mid+1)
    si array[mid]>mid entonces devolver GetMissingNumber(array,mid,low)
```

En esta solución el algoritmo está partiendo el vector a la mitad cada búsqueda que hace. Como esto puede hacerse $\mathcal{O}(log(n))$ veces, la complejidad resulta también $\mathcal{O}(log(n))$.

2.4. Análisis de las Soluciones

Mas allá del orden de complejidad de las tres soluciones, se pueden llegar a usar la primera ya que es muy sencilla de programar. Además se podría usar la primera solución si en vez de la estructura vector se emplea una lista enlazada. Si se elige las estructuras de datos vector se podría emplear la tercera solución con complejidad$\mathcal{O}(log(n))$. Como $n \leqslant m$ entonces $\mathcal{O}(log(n)) \subset \mathcal{O}(log(m))$

3. Un crédito sensible

Se propone un cuestionario de n preguntas. La respuesta a cada una de estas debe ser sólo “sí” o “no”. Además, cada cuestionario posee un único resultado posible: Aprobado o rechazado. Es posible representar a

cada cuestionario particular como una *n-úpla*. Sea $\{0,1\}^n$ el conjunto de todos los cuestionarios posibles de n preguntas. En el mismo tenor, se nota como $\{0,1\}$ al conjunto de resultados posibles para un cuestionario en particular. Con esto, se cuenta con una función $f : \{0,1\}^n \to \{0,1\}$. Definiendo a $S \subset \{1,\ldots,n\}$ como un subconjunto de índices y dado un cuestionario particular $x \in \{0,1\}$ se denota como x^S al cuestionario resultante de cambiar las respuestas indicadas por los índices en S en el cuestionario x. Por su parte, definimos a $\mathscr{S}$ como el conjunto de todos los S posibles para los cuales $f(x) \neq f(x^S)$. Esto es,

$$\mathscr{S} = \{S \in \mathcal{P}(\{1,\ldots,n\}) : f(x) \neq f(x^S)\}$$

Se define a la sensibilidad de un cuestionario particular x como la mínima cantidad de respuestas a cambiar para que el resultado de x cambie:

$$s(f,x) = \text{mín}\{|S| : S \in \mathscr{S}\} \tag{1}$$

Se define la sensibilidad global del cuestionario como el máximo para todas las sensibilidades de los cuestionarios particulares. Esto es,

$$s(f) = \text{máx}\{s(f,x) : x \in \{0,1\}^n\} \tag{2}$$

Con el fin de calcular $s(f)$ se analiza primero la opción por fuerza bruta. Esta es obtener $s(f,x)$ para todos los x posibles. Se sabe que existen 2^n cuestionarios particulares posibles. Para cada uno de estos, su sensibilidad puede estar, *a priori*, entre 1 y n.

Como cada cuestionario puede ser expresado por un número entero en binario es posible almacenar el resultado de estos en un vector v de 2^n elementos. A modo de ejemplo, para $n = 4$, el cuestionario $\{0,1,0,0\}$ se asocia al índice 0100_2, por lo que $v[0100_2] = v[4] = f(\{0,1,0,0\})$. Para minimizar los cambios de preguntas posibles en la pantalla se opta por utilizar el código de Gray. Esta sucesión permite recorrer todos los cuestionarios posibles cambiando sólo una respuesta por vez. Para $n = 3$ la sucesión resulta $000, 001, 011, 010, 110, 111, 101, 100$.

Con las consideraciones anteriores se utiliza el siguiente algoritmo:

Algoritmo 4 Cálculo de la sensibilidad global

```
v[0 : 2^n − 1] inicializado en −1
S ← 0
g(x) = Cantidad de apariciones de 1 en la representación binaria de x
para x ∈ {0, ..., 2^n − 1} hacer
    para x' ∈ {x ⊕ 0, x ⊕ 1, x ⊕ 11, x ⊕ 10, ...} hacer ▷ Código de Gray, representa, a lo sumo, un click
        si v[x'] = −1 entonces                          ▷ De no estar precalculado
            v[x'] ← f(x')                               ▷ Se calcula en O(f(n)), se ejecuta 2^n veces
        si v[x] ≠ v[x'] entonces
            S ← máx{S, g(x ⊕ x')}
devolver S
```

El algoritmo ocupa en memoria $\mathcal{O}(2^n)$, puesto que requiere un vector con exactamente 2^n elementos. Calcula $f(x)$ para todo x posible exactamente una vez. Por lo que es $\Omega(f(n)2^n)$. Finalmente, para cada elemento, analiza todos los restantes, esto es $\mathcal{O}(2^n 2^n) = \mathcal{O}(2^{2n}) = \mathcal{O}(4^n)$. Luego la complejidad resultante es $\mathcal{O}(4^n + f(n)2^n)$.

Se analiza un caso particular. En este caso un cuestionario cuenta con 20 preguntas, el algoritmo tarda 10 segundos en determinar si cada cuestionario particular fue aprobado, y cambiar una respuesta toma 0,4 segundos. En este caso se tienen $2^20 = 1048576$ cuestionarios posibles. Para cada cuestionario es necesario aplicar (según el algoritmo propuesto) una vez la función f. Esto toma 10485760 segundos. Cambiar entre estos cuestionarios toma 0,4 segundos por cada cuestionario (debido al uso del código de Gray), lo que representa 419430,4 segundos. Por lo que una cota inferior del tiempo que tomaría realizar este algoritmo es de 10485760 + 419430,4 segundos. Esta cota es de 126 días y no tiene en cuenta el tiempo que tome en ejecutarse el resto del algoritmo. Se aprecia que este tiempo no es razonable.

4. Trasplantes programados

Se cuentan con n pacientes y m donantes. Si bien puede resultar imposible conseguir un emparejamiento estable, puesto que de darse el caso $m \neq n$ no sería perfecto, es posible conseguir construir pares de pacientes–donantes de forma que estos sean estables.

Por lo que se considera como el mejor emparejamiento posible aquel que tenga mín$\{m, n\}$ parejas y además no presente inestabilidades.

Se define como preferencia de un paciente a la compatibilidad de este con un donante. Se define como preferencia de un donante a la factibilidad de este con un paciente.

Los donantes son representados con un número de 1 a m.

Los pacientes son representados con un número de 1 a n.

Las preferencias de cada donante–paciente son presentadas en un vector. Se utilizan dos formas distintas de representarlas. Para el caso del solicitante, el vector contiene a los requeridos en orden de preferencias. Esto es, el primer elemento del vector es el requerido al que el solicitante prefiere más, y el último al que prefiere menos. Para el caso del requerido, la preferencia del solicitante i se encuentra en la posición i del vector. Construir estos vectores es $\mathcal{O}(n \cdot m)$.

Con las consideraciones mencionadas, se presenta el algoritmo utilizado.

Algoritmo 5 Mejores emparejamientos entre donantes–pacientes

```
Sea S el conjunto de solicitantes
Sea R el conjunto de requeridos
M ← {}
para s ∈ S hacer
    se construyen preferencias de s
para r ∈ R hacer
    se construyen preferencias de r
mientras haya elementos en S hacer
    Se extrae s un elemento de S
    mientras s no esté emparejado y haya requeridos a los que todavía no les haya preguntado hacer
        Sea r el requerido de mayor preferencia al que s todavía no le preguntó
        si r no tiene pareja entonces
            Agregar (r, s) a M
        si no
            Sea s' el solicitante con el que r está emparejado
            si r prefiere a s más que a s' entonces
                Remover (r, s') de M
                Agregar s' a S
                Agregar (r, s) a M
devolver M
```

El algoritmo se ejecuta para cada solicitante y, en el peor caso posible, realiza una acción por cada requerido. Por lo que puede verse que es $\mathcal{O}(n \cdot m)$. Puesto que, o bien los solicitantes son m y los requeridos n, o bien es al revés. Definiendo $|S|$ como la cantidad de solicitantes y $|R|$ como la cantidad de requeridos, puede verse también que un solicitante nunca va a preguntarle a más requeridos que mín$\{|S|, |R|\}$. Esto es porque si $|S| < |R|$, los $|S| - 1$ solicitantes restantes sólo pueden estar asociados a $|S| - 1$ requeridos, luego como mucho debería preguntarle a $|S| - 1 + 1$ requeridos. Por otra parte si $|R| < |S|$, al preguntar a $|R|$ requeridos no podría preguntarle a ninguno más. La complejidad encontrada es, por lo tanto, $\mathcal{O}((\text{mín}\{|S|, |R|\})^2)$. Si bien esto sólo ocurre si se dispone de las preferencias previamente, de lo contrario la complejidad está limitada por la lectura de las preferencias y esta es $\mathcal{O}(n \cdot m)$.

Espacialmente el algoritmo debe guardar una tabla de preferencias por solicitantes y otra por requeridos. Cada tabla tiene $n \cdot m$ elementos. Además se requiere devolver mín$\{|S|, |R|\}$ parejas. Finalmente la complejidad espacial también resulta $\mathcal{O}(n \cdot m)$.

Este algoritmo ofrece emparejamientos perfectos cuando $n = m$. Como el emparejamiento resultante no presenta inestabilidades, además de perfecto, resulta un emparejamiento estable.

El algoritmo propuesto es conocido como algoritmo de Gale–Shapley y garantiza que los solicitantes obtengan su mejor pareja posible. Esto significa que dado un solicitante cualquiera, el requerido con el que

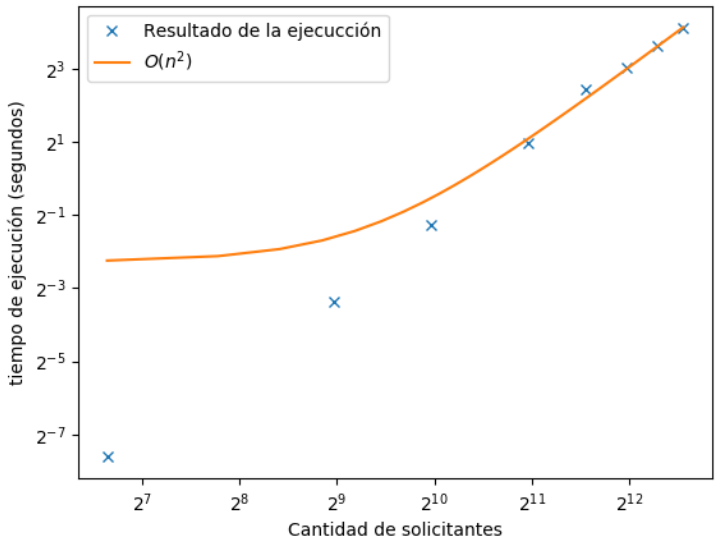

Figura 1: Tiempo de ejecución de la solución programada

está asociado en otra solución o bien es el mismo que en esta solución o bien es uno por el cual tiene menor preferencia. Es por eso que si los solicitantes son los pacientes, se garantizan las menores compatibilidades posibles (y por lo tanto los menores riesgos posibles) en cada donación. Pero se incurren en otros problemas como mayores gastos de transporte o donantes menos predispuestos.

Se programó una solución al problema y se analizó su complejidad resultando la misma que la teórica. Además, se crearon listas aleatorias de pacientes, donantes y preferencias y se comprobó para el caso $n = m$, que la complejidad resulta $\mathcal{O}(n^2)$. Esto puede verse en la figura 1.

Referencias

[1] Hao Huang. *Induced subgraphs of hypercubes and a proof of the Sensitivity Conjecture.* `https://arxiv.org/pdf/1907.00847v2.pdf`

[2] Jon Kleinberg, Éva Tardos *Algorithm Design* . Addison Wesley,2006.