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

(75.29 / 95.06) Teoría de Algoritmos

TRABAJO PRÁCTICO N.º 2
Grupo: "Los Fieles de Cardozo"
Reentrega: *13 de Noviembre 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 de cada consigna. La primera parte consiste en resolver un problema en el cual se usa teoria y algorimos de grafos, se expone la solución analisis y algoritmo utilizado. La segunda parte es un acercamiento a los algoritmos *Greedy*. Se analiza el algoritmo de compresión de datos *Huffman* y finalmente se programa. La ultima parte es optativa y consiste en analizar un *paper* donde se debe fundamentar si el algoritmo propuesto por *Donald B. Johnson* es tractable.

2. Despliegue de ayuda eficiente

El enunciado presenta un problema donde se debe minimizar la cantidad de equipos a armar cubriendo todo los centros urbanos existentes. Se considero que cada ciudad es el vértice o nodo de un grafo dirigido, las aristas son las vías de comunicación que siguen funcionando. Como entrada se reciben las aristas dirigidas del grafo.

La idea general de la solución es encontrar todas componentes fuertemente conexas del grafo principal para poder separarlo en sub-grafos fuertemente conexos. Aplicado al problema dado, al encontrar las componentes fuertemente conexas, se asegura que se puede partir de una ciudad dentro del sub-conjunto y volver a la misma recorriendo varias ciudades, entonces es posible optimizar la cantidad de equipos enviados.

Para poder obtener las componentes fuertemente conexas se decidió utilizar el algoritmo de *Kosaraju*. Este algoritmo consiste en recorrer el grafo con *Deep First Search (DFS)* mientras se van numerando los nodos recorridos hasta llegar a un nodo que no tenga adyacentes sin recorrer. Se podría utilizar la estructura de datos *Stack* para poder almacenar los nodos sin adyacentes primero y seguido seguir recorriendo hasta el final. Posterior a este primer paso se genera el grafo transverso invirtiendo las aristas y finalmente se va desapilando del stack los nodos. Todos los nodos adyacentes son componentes fuertemente conexas.

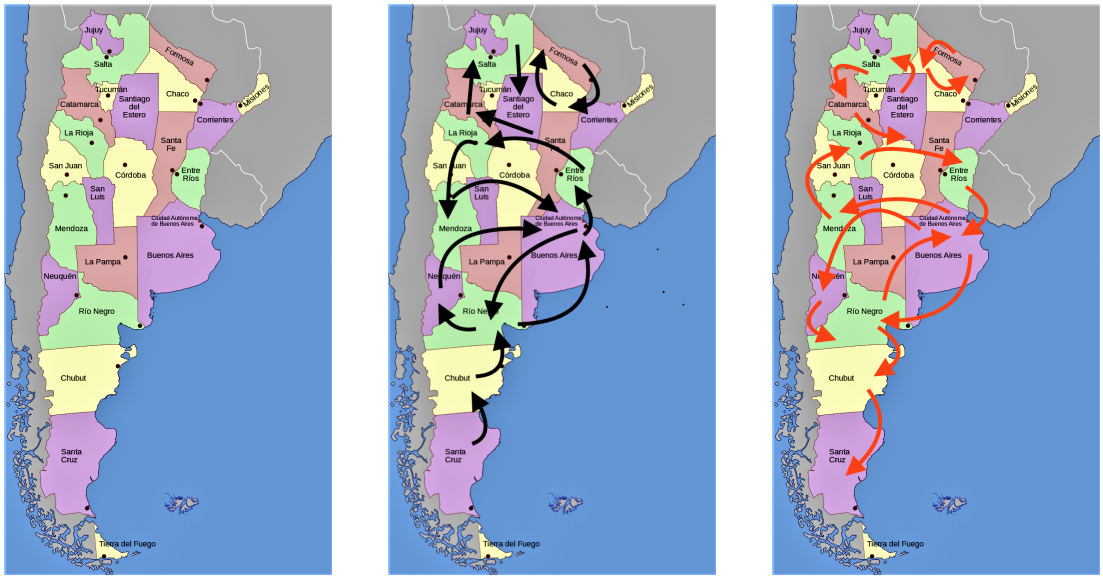
Algoritmo 1 Algoritmo de *Kosaraju*

```
inicializar c:=0
mientras todos los nodos no estén marcados hacer
    Elegir un nodo u
    hacer DFS(u)
    incrementar c
    marcar x como c
Girar las aristas del grafo (generar el grafo transverso)
para nodo u marcado de n hasta 1 hacer
    agrupar todos los nodos accesibles desde u y guardarlos como un conjunto
devolver Todos los conjuntos
```

Analizando la complejidad temporal de este algoritmo, siendo e la cantidad de aristas y v la cantidad de nodos empezamos por la función DFS es $\mathcal{O}(e+v)$. Crear el grafo transverso tiene una complejidad de $\mathcal{O}(e+v)$ ya que también hay que recorrer el grafo completo. Finalmente la complejidad del ciclo *for* final es análoga a llamar a DFS del grafo transverso. También tiene complejidad $\mathcal{O}(e+v)$. De esta forma la complejidad

total del algoritmo es $\mathcal{O}(e+v)$. La complejidad temporal del algoritmo se puede decir que es óptima, ya que es necesario recorrer todos los vértices y aristas para poder encontrar las componentes fuertemente conexas, seria imposible hacerlo sin poder recorrer algún vértice o arista ya que se tendría incompleta la información del grafo. Si se analiza la complejidad espacial, se obtiene $\mathcal{O}(v)$ ya que en el peor caso, debera guardar todos los vertices en el stack.

Para hacer un ejemplo visual, se puede imaginar que en las provincias de Argentina quedan dividas después del ataque extraterrestre, y las vías de comunicación que todavía quedan se pueden utilizar están dadas por las flechas al igual que un grafo dirigido.



(a) Mapa de las provincias de Argentina

(b) Mapa de las vías de comunicación que se pueden utilizar para transporte luego de la invasión

(c) Mapa de las vías transversas de comunicación

Figura 1: Mapas de la República Argentina

Las provincias que no tiene flecha son nodos sueltos que quedaron incomunicados con el resto de las provincias. Teniendo el cuenta el ejemplo de la figura 1 se puede visualizar el siguiente grafo

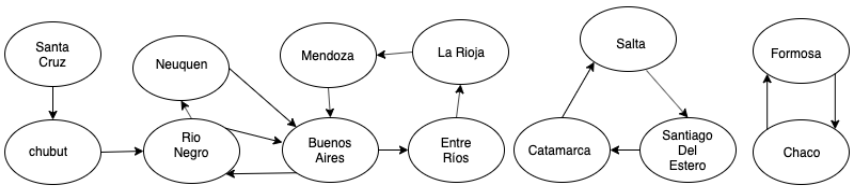


Figura 2: Esquema de grafo vías accesibles entre las provincias

Supongamos que empezamos en Santa Cruz con el algoritmo que encuentra las Componentes conexas. Se empieza haciendo *DFS* sobre el grafo, marca como visitado Santa Cruz y sigue recorriendo sus adyacentes hasta que llegue a uno marcado como visitado, en este caso Chubut, siguiendo por Rio Negro, Buenos Aires, Entre Rios, La Rioja, Mendoza. Al llegar a Mendoza como el proximo es Buenos Aires y este ya fue visitado se guarda a Mendoza en el stack y vuelve a La Rioja, como tampoco tiene mas adyacentes se lo guarda en el stack y asi tambien con entrerios. Al volver a Buenos Aires busca su adyacente y como vuelve a ser Rio Negro se guara Buenos Aires en el Stack y recorre el procimo adyacente a Rio Negro, en este caso Neuquen y al ser su adyacente ya visitado se guarda en el stack Rio Negro y al no tener mas Adyacentes Santa cruz se guarda en el Stack. Seguido recorre las otras componentes del grafo, si la siguiente provincia es chaco, se guardaran en el stack las provincias de Formosa y chaco. Finalmente se empieza por la provincia de Santiago del Estero se guardan el el stack las provincias de Salta, Catamarca y Santiago del Estero en orden.

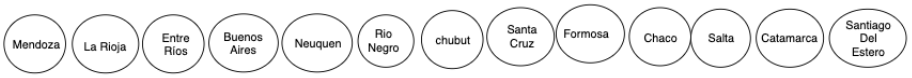


Figura 3: Esquema de la pila de nodos construida dentro del algoritmo de *Kosaraju* despues de recorrer todos los nodos y antes de recorrer el grafo transverso. El primer elemento en salir es el Santiago Del estero

Por ultimo, se construye el grafo transverso dando vuelta los vértices y se recorre desde los elementos que se traen del stack. En este caso se empieza obteniendo Santiago del Estero y se vuelve a marcar como vistado, se recorren sus adyacentes transversas, como solo tiene a Catamarca, se marca a Catamarca como visitado y como no tiene otro adyacente se agrupan como componente fuertemente conexa. Se vuelve a traer otro nodo del stack, como ya fue visitado se trae el siguiente nodo. Se recorren los adyacentes transversas de Salta, siguiendo por Chaco y Formosa, como formosa no tiene mas Adyacentes sin visitar se marcan las 3 provincias como visitadas y se agrupan como otra Componente fuertemente conexa. Se vuelve a retirar otro nodo del stack, Santa Cruz no tiene adyacentes, se marca como visitada, y se agrupa como otra componente fuertemente conexa, Se retira Chubut y se recorren sus adyacentes transversas.Como Santa Cruz ya fue visitada tambien se agrupa como una componente conexa. Finalmente se retira Rio Negro y vuelve a realizar el mismo proceso, dando como resultado las siguientes componentes fuertemente conexas del grafo estan encuadradas en la figura 4

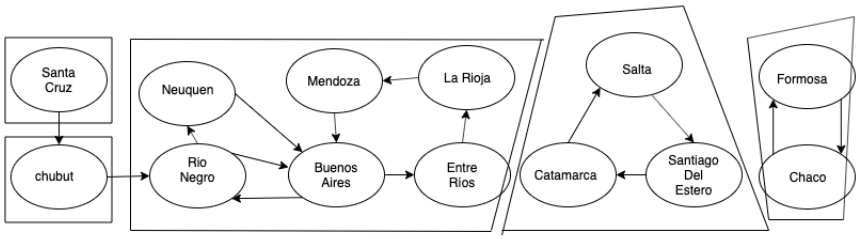


Figura 4: Esquema de las componentes fuertemente conexas del grafo de provincias

Otro uso practico que se le podría asociar a la búsqueda de componentes fuertemente conexas en un grafo es en la búsqueda de grupos de personas conocidas dentro de una red social. Si se considera que cada perfil personal en una red social es un vértice y la relación (sea de amistad, conectado o seguidor) es una arista, las componentes fuertemente conexas en este caso seria cada grupo de personas que se conocen entre si y se podría establecer cierta relación real entre los integrantes de este grupo.

3. Comunicación satélital

Como parte del programa espacial argentino se pidió programar el algoritmo de compresión de datos *Huffman* para utilizar en la transmisión de las imágenes enviadas desde los satélites.

Un algoritmo *Greedy* es una estrategia de solución que consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. Una vez llegada a la solución se demuestra que es óptima o que tiene cierta cercanía con un óptimo propuesto.

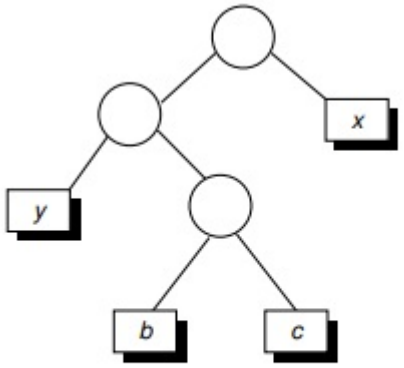
El algoritmo de *Huffman* construye un árbol teniendo en cuenta la frecuencia de cada caracter. En cada paso combina símbolos de menor frecuencia en un nuevo símbolo cuya frecuencia puede expresarse como la suma de estos. Este proceso se hace recursivamente hasta obtener el nodo raíz cuyo símbolo aúna a todos los caracteres. De esta manera se consigue con código sin prefijos.

Demostración: Sea T un árbol binario de codificación prefija óptimo. Sean b y c dos hojas hermanas en T que se encuentran a profundidad máxima. Sean x e y dos hojas de T tales que son los 2 caracteres del alfabeto C con la frecuencia más baja. El caso interesante para la demostración se produce cuando $(b,c) \neq (x,y)$.

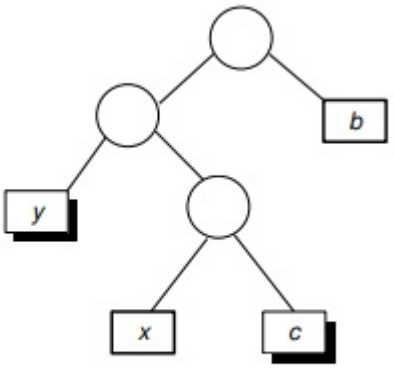
T, que es un árbol óptimo, se puede transformar en otro árbol $T"$, también óptimo, en el que los 2 caracteres, x e y, con la frecuencia más baja serán hojas hermanas que estarán a la máxima profundidad. El árbol que genera el algoritmo cumple exactamente esa condición. Podemos suponer que $f(b) \leq f(c)$ y que $f(x) \leq f(y)$.

También se puede deducir que $f(x) \leq f(b)$ y $f(y) \leq f(c)$.

Construimos un nuevo árbol, T', en el que se intercambia la posición que ocupan en T las hojas b y c.



árbol T



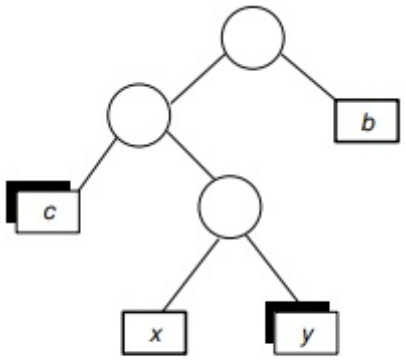
árbol T'

El número de bits para el nuevo árbol de codificación T' lo denotamos por $B(T')$.

Tenemos entonces que: $B(T)-B(T') = \sum_{c\in C} f(c)\cdot d_T(c) - \sum_{c\in C} f(c)\cdot d_{T'}(c) = f(x)\cdot d_T(x)+f(b)\cdot d_T(b)-f(x)\cdot d_{T'}(x)-f(b)\cdot d_{T'}(b) = f(x)\cdot d_T(x)+f(b)\cdot d_T(b)-f(x)\cdot d_T(x)-f(b)\cdot d_T(b) = (f(b)-f(x))\cdot(d_T(b)-d_T(x)) \geq 0$.

De modo que $B(T) \geq B(T')$ pero dado que T es óptimo, mínimo, T' no puede ser menor y $B(T) = B(T')$ por lo que T' también es óptimo.

De forma similar se construye el árbol T'' intercambiando c e y.



árbol T''

Con este intercambio tampoco se incrementa el coste $B(T') - B(T'') \geq 0$. Por tanto, $B(T'') = B(T)$ y como T es óptimo, entonces T'' también lo es.

Algoritmo 2 Construcción de un árbol de Huffman

Sea `heap` un heap de mínimo vacío
para cada caracter c cuya frecuencia f **hacer**
    insertar en `heap` nodo de etiqueta c y peso f
mientras `heap` tenga más de un elemento **hacer**
    árbol_a ← extraer(`heap`)
    árbol_b ← extraer(`heap`)
    etiqueta ← Concatenación de las etiquetas de los nodos raíz de árbol_a y árbol_b
    peso ← peso(nodo raíz de árbol_a) + peso(nodo raíz de árbol_b)
    árbol_c ← árbol de etiqueta `etiqueta`, peso `peso` y subárboles izquierdo y derecho `árbol_a` y `árbol_b` respectivamente
    insertar árbol_c en `heap`
árbol ← extraer(`heap`)
devolver árbol

Se detalla en el algoritmo 2 la construcción un árbol de Huffman. Sea n la cantidad de caracteres a

utilizar. La construcción de un *heap* de n elementos es de orden $\mathcal{O}(n \log n)$. En cada paso se extraen dos árboles parciales del *heap* y se construye un nuevo árbol a partir de ellos, uniendo sus etiquetas. Ambas operaciones son de orden $\mathcal{O}(\log n)$. Este nuevo árbol es insertado en el *heap*, esta inserción es $\mathcal{O}(\log n)$. Se puede ver que esta operación se llevará a cabo $\mathcal{O}(n)$ veces. Para ello se considera n caracteres separados por espacios en el orden en el que aparecen en la etiqueta del nodo raíz al finalizar el algoritmo. Es posible representar a una unión como eliminar un espacio en esta cadena. Para unir todos los caracteres es necesario eliminar $n-1$ espacios presentes. Por ende, la operación de unión se lleva a cabo exactamente $n-1$ veces, y esto es $\mathcal{O}(n)$. Además, las operaciones de mayor complejidad algorítmica del bucle son la inserción y extracción del *heap*, ambas de orden $\mathcal{O}(\log n)$. Luego el algoritmo es de orden $\mathcal{O}(n \log n)$.

El algoritmo 3 detalla la compresión de un una cadena utilizando el árbol creado. Se construye un vector de 256 posiciones que sirva para asociar cada byte posible a su cadena de bits correspondiente al comprimirse. Esto puede realizarse en $\mathcal{O}(n \log n)$. Para ver que esto es así se toman dos casos extremos. Si todas las frecuencias fueran iguales, el árbol de Huffman resultante es un árbol completo y recorrer todos sus nodos es $\mathcal{O}(n \log n)$. De forma opuesta, pueden proponerse frecuencias para que el árbol de Huffman tenga altura $\mathcal{O}(n)$. Un ejemplo de esto resulta de tomar frecuencias que sigan la sucesión de Fibonacci. Sin embargo, al ser la altura de un subárbol $\mathcal{O}(n)$ la del subárbol hermano debe ser $\mathcal{O}(1)$. Las alturas de cada subárbol están relacionadas. Todo árbol de Huffman se encuentra entre estos dos casos (el de un árbol completo o el de un árbol degenerado en una lista). Como recorrer el árbol degenerado en una lista es $\mathcal{O}(n)$, y recorrer el completo es $\mathcal{O}(n \log n)$ se concluye que recorrer todo árbol de Huffman tendrá una complejidad entre estas dos. Finalmente en el peor caso esto es $\mathcal{O}(n \log n)$. Por lo que construir el vector que permite mapear las claves es una operación de complejidad $\mathcal{O}(n \log n)$. Comprimir cada byte individualmente es obtener un elemento del vector, esto es $\mathcal{O}(1)$. Luego, la cadena de entrada tiene m bytes, el orden de complejidad temporal del algoritmo de compresión resulta $\mathcal{O}(m \cdot n \log n)$.

Algoritmo 3 Compresión de un archivo

```
Se construye HT el árbol de Huffman con las frecuencias dadas
Sea v un vector tal que para cada byte b posible v[b] sea el byte b comprimido según HT
bits ← arreglo de bits vacío
para cada byte b en el archivo hacer
    Agregar v[b] a bits
devolver bits
```

Un algoritmo de descompresión se detalla en el algoritmo 4. Sea n la cantidad de bits en el archivo que se desea descomprimir. Para cada bit se realiza una operación de orden de complejidad $\mathcal{O}(1)$. Luego la descompresión es de orden $\mathcal{O}(n)$.

Algoritmo 4 Descompresión de un archivo

```
Sea HT el árbol de Huffman cuyo código fue usado para comprimir el archivo
descomprimido ← una cadena vacía
árbol_actual ← HT
mientras cada bit b en el archivo hacer
    si b es 0 entonces
        árbol_actual ← subárbol izquierdo de árbol_actual
    si no
        árbol_actual ← subárbol derecho de árbol_actual
    si árbol_actual es un nodo raíz entonces
        agregar la etiqueta de árbol_actual a descomprimido
        árbol_actual ← HT
devolver descomprimido
```

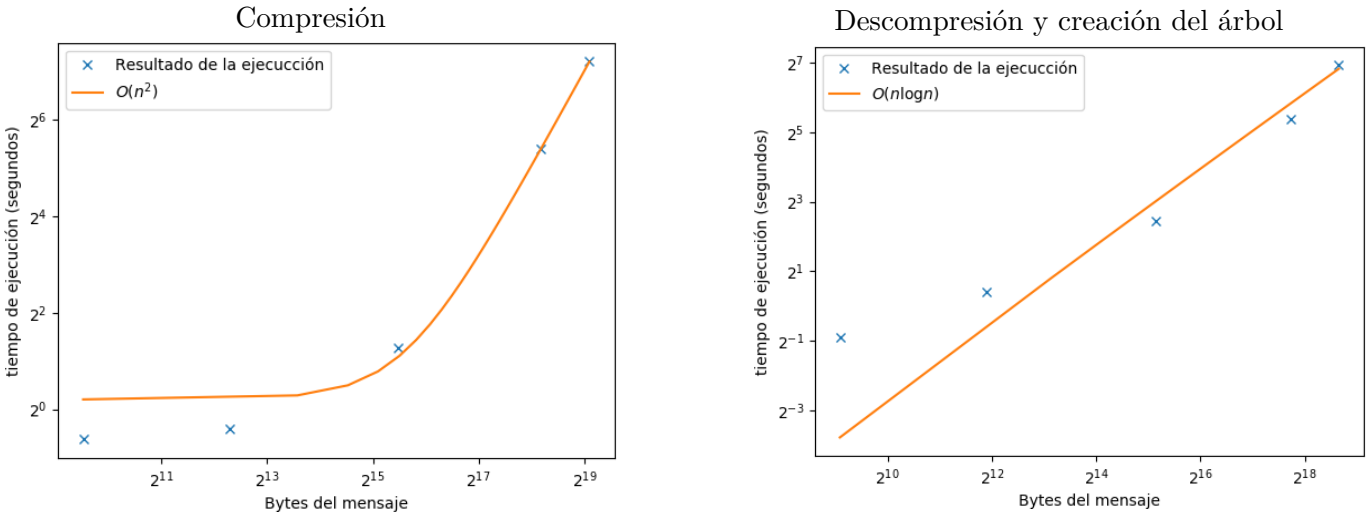


Figura 5: Resultado de los datos relevados.

En la figura 5 se aprecia como si bien la complejidad resultante para el algoritmo de compresión desarrollado es la teórica en el caso de la descompresión esto no es así. Se atribuye la discrepancia a la implementación del arreglo de bits utilizado. En este, las operaciones de inserción y extracción resultan $\mathcal{O}(n)$ en el peor caso.

4. ¿Es el siguiente un algoritmo tractable?

En esta tercera parte se analizó el algoritmo de *Johnson*, *"Finding All The Elementary Circuits Of A Directed Graph"* y se decidió si estamos frente a un algoritmo tractable o no. El orden de complejidad de este algoritmo es $\mathcal{O}((n+e)(c+1))$ donde n es la cantidad de nodos, e es la cantidad de aristas y c es la cantidad de ciclos que tiene el grafo. Si se analiza el peor caso, donde tenemos un grafo dirigido completo de n nodos, se tienen $e = n(n-1)$ aristas. La cantidad de todos los subconjuntos formados por los nodos es su cardinal, en este caso 2^n.Como para formar un ciclo se necesitan a lo sumo 2 nodos, para obtener la cantidad de conjuntos que se pueden formar con 2 o mas nodos es la cantidad total restado la cantidad de conjuntos que tiene 1 solo elemento, en este caso $2^n - n$ por esta razon la cantidad de ciclos que puede tener un grafo completo es $c = 2^n - n$ ciclos. En este peor caso la complejidad algoritmica total sería

$$\mathcal{O}((n^2)(2^n - n + 1)) = \mathcal{O}((n^2 \cdot 2^n - n^3 + n^2)) = \mathcal{O}((n^2 \cdot 2^n))$$

Siendo este complejidad no polinomial no sería tractable. Se puede llegar a decir que este algoritmo pertenece a los problemas Pseudo polinomiales o NP-completos debiles ya que dependiendo de la la cantidad de nodos, aristas y forma del grafo podría tener una complejidad polinomial.

Referencias

[1] Donald B. Johnson. *"Finding All The Elementary Circuits Of A Directed Graph"*. `https://www.cs.tufts.edu/comp/150GA/homeworks/hw1/Johnson%2075.PDF`

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