Diseño de Algoritmos - Algoritmos II

(notas basadas en el material adicional de "The Design & Analysis of Algoritmos" (Levitin 2003))

Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Clase 4: La Estrategia Decrease & Conquer

Decrease & Conquer

La estrategia de diseño de algoritmos Decrease & Conquer apunta a resolver problemas algorítmicamente de la siguiente forma:

Reducir instancias del problema a instancias más pequeñas del mismo problema

Resolver las instancias más pequeñas

Extender las soluciones de instancias más pequeñas para conseguir una solución a la instancia original

Tipos de Decrease & Conquer

De acuerdo a la forma de "decrementar" un problema, existen diferentes tipos de Decrease & Conquer

- **Decremento por una constante** (usualmente 1):
  - insertion sort / selection sort
  - algoritmos de recorrido de grafos (DFS and BFS)
  - ordenamiento topológico
  - algoritmos para generar permutaciones, subconjuntos, ...
- **Decremento por un factor constante** (usualmente la mitad)
  - búsqueda binaria
  - exponenciación por cuadrados
- **Decremento de tamaño variable**
  - Algoritmo de Euclides

Un Ejemplo: Insertion Sort

Para ordenar un arreglo A[0..n-1], ordenar primero A[0..n-2] y luego insertar A[n-1] en el lugar que le corresponde dentro del segmento ya ordenado A[0..n-2]

Ejemplo: ordenar 6, 4, 1, 8, 5

6 | 4 1 8 5

4 6 | 1 8 5

1 4 6 | 8 5

1 4 6 8 | 5

1 4 5 6 8

Pseudo-código de Insertion Sort

ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```


Análisis de Insertion Sort

- Eficiencia en tiempo

$C_{worst}(n) = n(n-1)/2 \in \Theta(n^2)$

$C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

$C_{best}(n) = n - 1 \in \Theta(n)$ (es rápido para arreglos "casi ordenados")

- Eficiencia en Espacio: in-place
- Estable? Si
- En líneas generales, es el mejor algoritmo de ordenamiento "básico"

Recorridos en Grafos

Muchos problemas requieren representar datos como grafos (relaciones), y utilizan recorridos de grafos para procesar información de forma sistemática

Algoritmos básicos de recorrido de grafos:

- Depth-first search (DFS)
- Breadth-first search (BFS)

Depth-First Search (DFS)

Visita los vértices del grafo moviéndose desde el último vértice visitado hacia uno no visitado, y dando marcha atrás (backtracking) si no hay vértices no visitados disponibles.

Para su implementación, se puede utilizar una pila

un vértice se apila cuando se alcanza por primera vez

un vértice se quita de la pila cuando se "agota", es decir cuando ya no existen vértices no visitados adyacentes al mismo

Depth First Search genera un "árbol" de visita a partir de un grafo

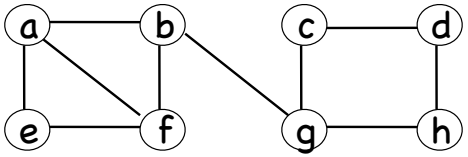
Pseudo-código de DFS

ALGORITHM *DFS*(G)

```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable count
count ← count + 1; mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```


Ejemplo: DFS sobre grafo no dirigido



Pila DFS:

Árbol DFS:

Características de DFS

- Eficiencia de acuerdo a representación de grafos:
  - matrices de adyacencia: $\Theta(V^2)$
  - listas de adyacencia: $\Theta(|V|+|E|)$
- Da lugar a diferentes ordenamientos de vértices:
  - orden en que los vértices son visitados
  - orden en el cual los vértices se "agotan"
- Aplicaciones:
  - chequeo de conectividad, búsqueda de componentes conexas
  - chequeo de aciclicidad
  - búsqueda de soluciones en espacios de búsqueda de problemas (IA)

Breadth-first search (BFS)

- Visita los vértices de un grafo moviéndose entre todos los vecinos del último vértice visitado
- Puede implementarse de manera similar a DFS, usando una cola en lugar de una pila
- Similar a la visita a un árbol por niveles
- Produce a partir de un grafo una estructura en forma de árbol

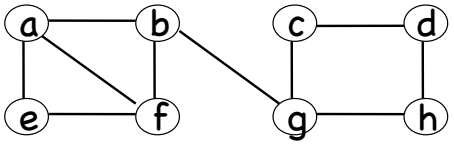
Pseudo-código de BFS

ALGORITHM *BFS*(G)

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = (V, E)
//Output: Graph G with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v by a path
//and assigns them the numbers in the order they are visited
//via global variable count
count ← count + 1;  mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;  mark w with count
            add w to the queue
    remove the front vertex from the queue
```


Ejemplo de Recorrido BFS sobre un Grato no Dirigido



Cola de recorrido BFS:

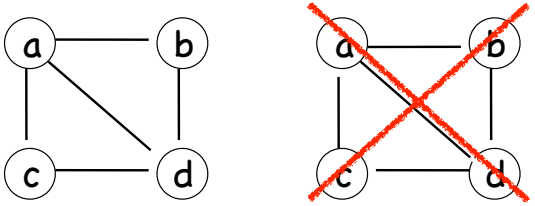
Árbol BFS

Características de BFS

- BFS tiene la misma eficiencia que DFS:
  - matrices de adyacencia: $\Theta(|V|^2)$
  - listas de adyacencias: $\Theta(|V|+|E|)$
- Produce un único orden entre los vértices del grafo (el orden en que los vértices se agregan y quitan de la cola es el mismo)
- Aplicaciones: Las mismas que DFS; permite también encontrar los caminos de un nodo a otros con número mínimo de arcos

DAGs y Ordenamiento Topológico

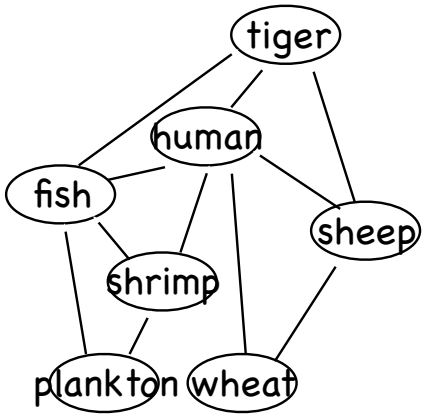
Dag: grafo dirigido acíclico (directed acyclic graph), i.e., un grafo dirigido sin ciclos (dirigidos)



Surgen en el modelado de muchos problemas que involucran la noción de pre-requisito (e.g., control de versiones)

Los vértices de un dag pueden ordenarse de manera tal que para cada arco su vértice de origen se liste antes que su vértice destino (***ordenamiento topológico***). El grafo debe ser un dag para que este proceso sea posible.

Ejemplo de Ordenamiento Topológico



Ordenar los siguientes elementos en la cadena alimenticia.

Algoritmo Basado en DFS

Algoritmo basado en DFS para Ordenamiento Topológico

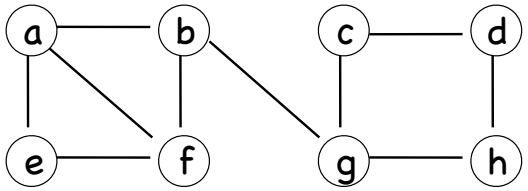
Realizar un recorrido en DFS, "anotando" el orden en que los vértices se quitan de la pila de recorrido

El orden inverso es un orden topológico

Si se encuentran arcos de retorno, no es un DAG

Eficiencia?

Ejemplo:



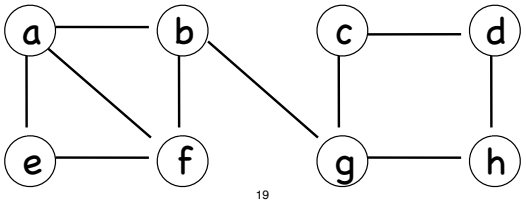
Algoritmo de Eliminación de Orígenes

Algoritmo de Eliminación de Orígenes

Iterativamete, identificar y eliminar los "orígenes" (vértices sin arcos entrantes), y todos los arcos que de ellos salen, hasta que ya no queden vértices (problema resuelto) o hasta que haya vértices pero no orígenes (no es un dag)

Eficiencia: La misma que para el algoritmo basado en DFS.

Ejemplo:



Algoritmos de Decremento por Factor Constante

En esta variante de Decrease & Conquer, el tamaño de instancia de problema se reduce en un mismo factor en cada caso (usualmente, 2)

Ejemplos:

Búsqueda binaria

Exponenciación mediante cuadrados

Problema de la moneda falsa

Exponenciación por Cuadrados

El Problema: Computar a^n donde n es un entero no negativo

El problema se puede resolver aplicando recursivamente las siguientes fórmulas:

Para valores pares de n $\quad a^n = (a^{n/2})^2$ si $n > 0$ y $a^0 = 1$

Para valores impares de n $\quad a^n = (a^{(n-1)/2})^2 a$

Recurrencia: $M(n) = M(\lfloor n/2 \rfloor) + f(n)$, donde $f(n) = 1$ o 2,
$M(0) = 0$

Teorema Maestro: $M(n) \in \Theta(\log n)$

Multiplicación por Duplicación

El problema: Computar el producto de dos enteros positivos

Se puede resolver mediante un decremento a la mitad, usando las siguientes fórmulas:

Para valores pares de n: $\quad n * m = \frac{n}{2} * 2m$

Para valores impares de n:

$$n * m = \frac{n-1}{2} * 2m + m \text{ si } n > 1 \text{ y } m \text{ si } n = 1$$

Algoritmos de Decremento Variable

En esta variante de Decrease & Conquer se reduce cada instancia del problema en tamaños variables de un paso a otro

Ejemplos:

- Algoritmo de Euclides
- Búsqueda por Interpolación
- Algoritmos sobre árboles binarios de búsqueda (no balanceados)

Algoritmo de Euclides

El Algoritmo de Euclides se basa en la aplicación iterativa de la siguiente igualdad

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

Ej.: gcd(80,44) = gcd(44,36) = gcd(36, 12) = gcd(12,0) = 12

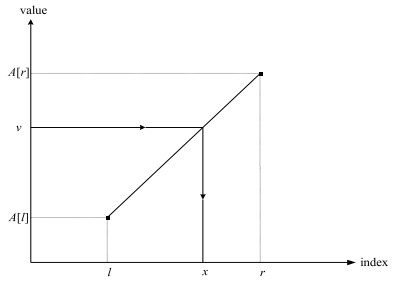
Se puede demostrar que el tamaño, medido como el segundo valor, decrece al menos a la mitad en dos iteraciones consecutivas.

Luego, $T(n) \in O(\log n)$

Búsqueda por Interpolación

Es una búsqueda en una secuencia ordenada A[l..r], similar a la búsqueda binaria, pero estimando la ubicación de la clave buscada en el segmento a explorar, usando su valor.

Más precisamente, se supone que los valores del arreglo crecen linealmente, y la ubicación de la clave v a buscar se estima como el valor de la coordenada x del punto correspondiente a v, en el segmento que une (l, A[l]) con (r, A[r]), y cuya coordenada x es v:



$$x = l + \lfloor (v - A[l])(r - l)/(A[r] - A[l]) \rfloor$$

Análisis de la Búsqueda por Interpolación

- Eficiencia

  caso promedio: $C(n) < \log_2 \log_2 n + 1$

  peor caso: $C(n) = n$

- Es preferible a la búsqueda binaria sólo para búsquedas sobre espacios muy grandes, o donde las comparaciones son costosas