# Análisis y Diseño de Algoritmos.

**Sesión 8.** 9 de Octubre de 2014.

Maestría en Sistemas Computacionales.

Por: Hugo Iván Piza Dávila.

# Algoritmos Voraces

- Utilizados para resolver problemas de optimización:
  1. Ejecutar todas las tareas pendientes siguiendo una secuencia que minimice el tiempo total.
  2. Encontrar el camino más rápido/barato desde A hasta B.
  3. Introducir objetos de valor en una mochila tal que el valor de los objetos introducidos sea el máximo posible y sin que se supere la capacidad de la mochila.
  4. Encontrar una ruta que visite a todas las ciudades y regresa al punto de inicio en el menor tiempo posible.
  5. Comunicar todos los puntos de una red utilizando la menor cantidad de cable.

# Algoritmos Voraces

- Se caracterizan por tomar la mejor decisión local, sin una visión global del problema.
  - Por tanto, no garantizan optimización global.
- Son más fáciles de programar que sus competidores.
- Cuando *funcionan* (sí entregan una solución óptima) son mejores que sus competidores porque son más eficientes.
  - Esto significa que hay problemas que no pueden ser resueltos de manera óptima por los Algoritmos Voraces.
- Nos enfocaremos en los casos que sí *funcionan*.

# ¿Por qué no siempre funcionan?

- Analicemos un caso trivial: *el problema del cambio*.
- "Dado un conjunto M de monedas de diferente denominación, y una cantidad monetaria C a entregar como cambio, entregar la cantidad exacta con la mínima cantidad de monedas posible".
- De acuerdo a las denominaciones existentes en nuestro sistema monetario (1, 2, 5, 10, 20) el algoritmo voraz siempre da una solución óptima:
  - En cada paso se elige la moneda de mayor denominación posible tal que la suma de las monedas elegidas no supere el cambio.

# ¿Por qué no siempre funcionan?

- Desafortunadamente, los problemas computables reales no son así de simples.
- Para añadirle peligro, estas son las monedas disponibles: M = {1, 1, 1, 1, 4, 4, 4, 6, 6, 10, 10} y el cambio C = 9.
- Un algoritmo voraz me entregaría la solución: {6, 1, 1, 1}.
  - Elige la moneda más grande disponible (información local) que siempre estará en la solución. No considera otras combinaciones.
  - Cada moneda descartada, nunca será reconsiderada.
- Sin embargo, la mejor solución es: {4, 4, 1}.
  - $|\{4, 4, 1\}| = 3 < 4 = |\{6, 1, 1, 1\}|$

# Características de un A. V.

1. Se cuenta con un conjunto (o lista) de candidatos.
   - Las monedas disponibles.
   - Los objetos de valor a introducir en la mochila.
   - Las ciudades a visitar.
   - Las tareas que se planean ejecutar.
2. En la ejecución, otros dos conjuntos comienzan a crecer:
   a) Candidatos que han sido considerados y elegidos.
   b) Candidatos que han sido considerados y rechazados.

# Características de un A. V.

3. Existe una función que verifica si los candidatos elegidos resuelven el problema.
   - La suma de las monedas elegidas es igual al cambio.
   - La mochila ya no tiene capacidad para otro objeto más.
   - Se visitaron todas las ciudades / ya se llegó al punto B.
   - Se programaron todas las tareas.
4. Existe otra función que revisa si el conjunto de candidatos es **factible** o no. *No siempre es necesario*.
   - Con las monedas elegidas ya superamos el cambio, o las monedas disponibles no son suficientes para llegar al cambio.
   - Con la ruta actual, el punto B es inalcanzable.

# Características de un A. V.

5. Existe una función de ***Selección*** que determina en cualquier momento cuál es el candidato más promisorio de los que aún no se han elegido.
   - La moneda de mayor denominación que no supere el cambio.
   - El objeto de mayor valor que no supere la capacidad de la mochila.
   - La ciudad más cercana a la última de la ruta actual.
6. Finalmente, existe una función ***Objetivo*** que calcula el valor de la solución encontrada (lo que se desea maximizar/optimizar):
   - El número de monedas utilizadas para dar el cambio.
   - El tiempo de ejecución de la secuencia de tareas.
   - La cantidad de cable que se utilizó para unir todos los puntos.

# Algoritmo general

```
Set/List AlgoritmoVoraz(Set/List C) {
    Sea S ← ∅  {La solución se guardará en S}
    Mientras C ≠ ∅ y Solución(S), hacer:
        Sea x ← Seleccionar(C)
        C ← C \ {x}
        Si Factible(S ∪ {x}) entonces S ← S ∪ {x}
    Fin-Mientras
    Si Solución(S) devolver S
    si no, devolver ∅ {No hay soluciones}
}
```

# ¿Cuáles problemas resolveremos?

- De los problemas de optimización ejemplificados, existen tres que sí son resueltos por un algoritmo voraz:
  1. El problema de la mochila si podemos partir los objetos.
  2. Comunicar todos los puntos de una red utilizando la menor cantidad de cable.
  3. Encontrar el camino más rápido/barato desde A hasta B.
- Por lo tanto, son los que resolveremos (en ese orden).

# Problema de la Mochila

- Existen $N$ objetos disponibles y una mochila.
- Cada objeto $k$ pesa $w_k$ y vale $v_k$, tal que $w_k, v_k > 0$.
- La mochila puede cargar un paso no mayor a $W$.
- En esta versión podemos meter a la mochila una fracción $f_k$ del objeto $k$, tal que: $0 \leq f_k \leq 1$.
- El problema se define formalmente así:
  - $Maximizar \ \sum_{k=1}^{N} f_k v_k$ sujeto a: $\sum_{k=1}^{N} f_k w_k \leq W$

# Problema de la Mochila

```
List KnapSack(List-of-items C, Max-weight W) {
     List-of-items S ← ∅
     weight ← 0
     Mientras weight < W, hacer:
          Sea k ← Seleccionar(C) {Devuelve el índice}
          Sea x ← C[k] {Obtiene el elemento}
          C ← C \ {x}
          Si weight + x.weight ≤ W    {Aún cabe, tomamos todo el objeto}
               x.fraction = 1.0
               weight ← weight + x.weight
          Si no  {Ya rebasamos W, tomamos la máxima fracción del objeto}
               w.fraction = (W – weight) / x.weight
               weight = W
          S ← S ∪ {x}
     Fin-Mientras
     Devolver S
}
```

# Problema de la Mochila

- ¿Cuál elemento seleccionamos?
  1. ¿El más valioso?
  2. ¿El más ligero?
  3. ¿El más valioso por unidad de peso?
- Veamos la siguiente instancia del problema de la mochila:

| $W = 100$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $v$ | 20 | 30 | 66 | 40 | 60 |
| $w$ | 10 | 20 | 30 | 40 | 50 |
| $v / w$ | 2.0 | 1.5 | 2.2 | 1.0 | 1.2 |

# Problema de la Mochila

| $W = 100$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $v$ | 20 | 30 | 66 | 40 | 60 |
| $w$ | 10 | 20 | 30 | 40 | 50 |
| $v / w$ | 2.0 | 1.5 | 2.2 | 1.0 | 1.2 |

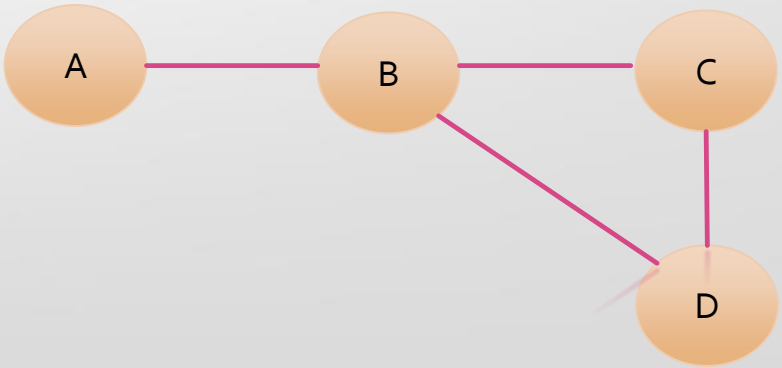
| *Seleccionar* | 1 | 2 | 3 | 4 | 5 | Valor |
|---|---|---|---|---|---|---|
| *Max* $v_k$ | 0 | 0 | 1 | 0.5 | 1 | 146 |
| *Min* $w_k$ | 1 | 1 | 1 | 1 | 0 | 156 |
| *Max* $v_k / w_k$ | 1 | 1 | 1 | 0 | 0.8 | 164 |

# Grafos

- Muchos problemas computables pueden ser formulados en términos de objetos y conexiones entre ellos:
  - Dado un mapa de rutas aéreas de un país, ¿cuál es el camino más rápido de llegar de una ciudad a otra? ¿cuál es el camino más barato de llegar de una ciudad a otra?
  - Dado un circuito eléctrico compuesto de transistores, resistores y capacitores, ¿todos los elementos están conectados? ¿funcionará correctamente?
  - Dado un conjunto de tareas por realizar que son activadas con la terminación de otra(s), ¿en qué momento comienza cada tarea?
  - Dada una red social, ¿existen grupos de usuarios que no estén relacionados con ningún otro contacto de la red? ¿existe algún usuario que pueda ser conocido por otro que no es su contacto?

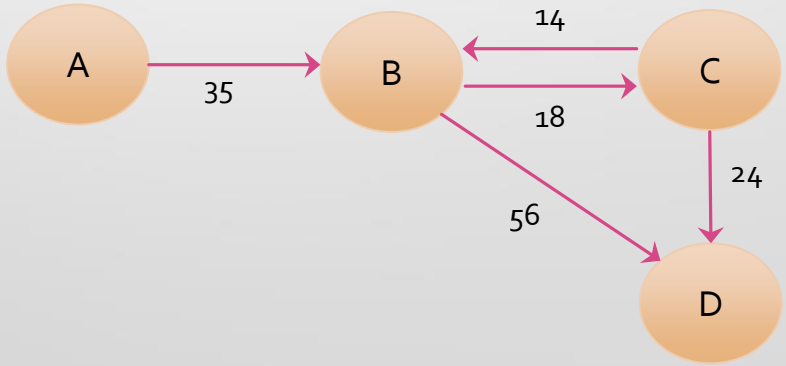
# Grafos

- Desde el punto de vista de programación, un Grafo es una estructura de datos compuesta por *nodos* interconectados a través de *aristas*
- Cada nodo está identificado por un valor.
- El siguiente grafo tiene cuatro nodos {A, B, C, D} y cuatro aristas {(A,B),(B,C),(B,D),(C,D)}.
  - Nótese que: (A,B) = (B,A)



# Tipos de grafos

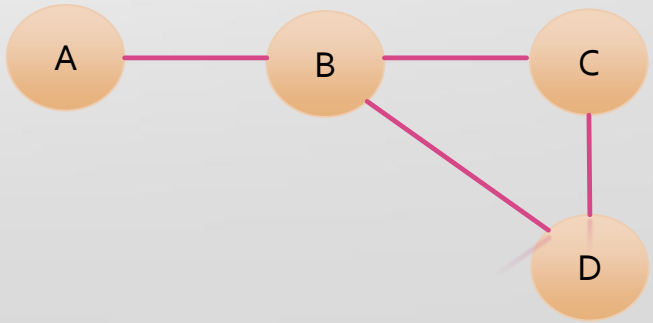
- Existen grafos ponderados, en los cuales se asignan valores o pesos a las aristas (distancia, costo, tiempo,…)
- Existen grafos dirigidos, en los cuales las aristas tienen un sentido: (A, B) no implica (B, A)
- Existen grafos ponderados dirigidos o redes, que combinan las dos características anteriores:



# Representación

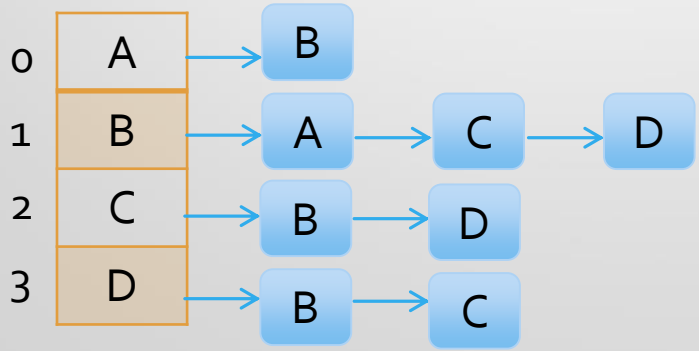
- Matrices de adyacencia.
  - Fácil implementación. Eficiente en el acceso.
  - Apropiado para Grafos Completos o Densos.
    - Grafo completo: el número de aristas es el máximo posible: ½ N (N – 1)
    - El número de aristas es mayor que N log N. (N = número de vértices).
  - Para Grafos no Densos habría mucho desperdicio.
  - No permite crecimiento en el número de vértices.

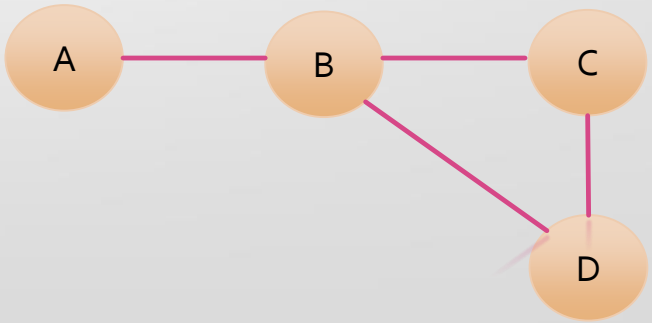
| | | | |
|---|---|---|---|
| 0 | **1** | **0** | **0** |
| 1 | 0 | **1** | **1** |
| 0 | 1 | 0 | **1** |
| 0 | 1 | 1 | 0 |



# Representación

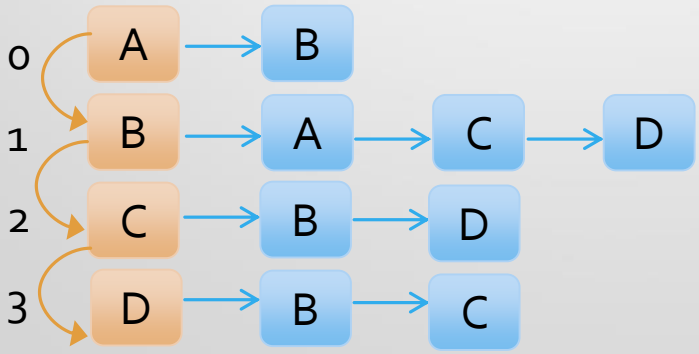
- Listas de adyacencia.
  - Implementación menos fácil.
  - Menos desperdicio de memoria en Grafos no Densos.
  - Permite crecimiento en número de aristas.
  - El recorrido en profundidad tiene complejidad lineal.
  - No permite crecimiento en número de vértices.

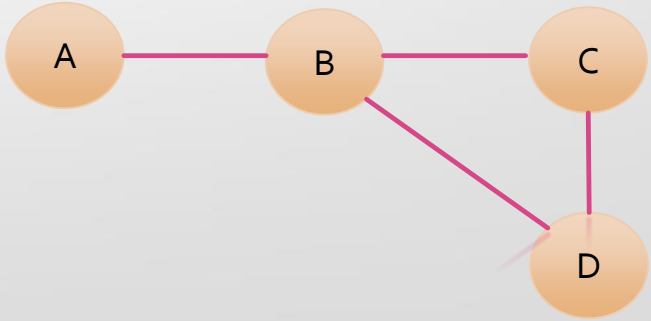


# Representación

- Listas dinámicas de adyacencia.
  - Implementación aún menos fácil.
  - Menos desperdicio de memoria en Grafos no Densos.
  - El recorrido en profundidad tiene complejidad lineal.
  - Permite crecimiento en número de vértices y aristas.



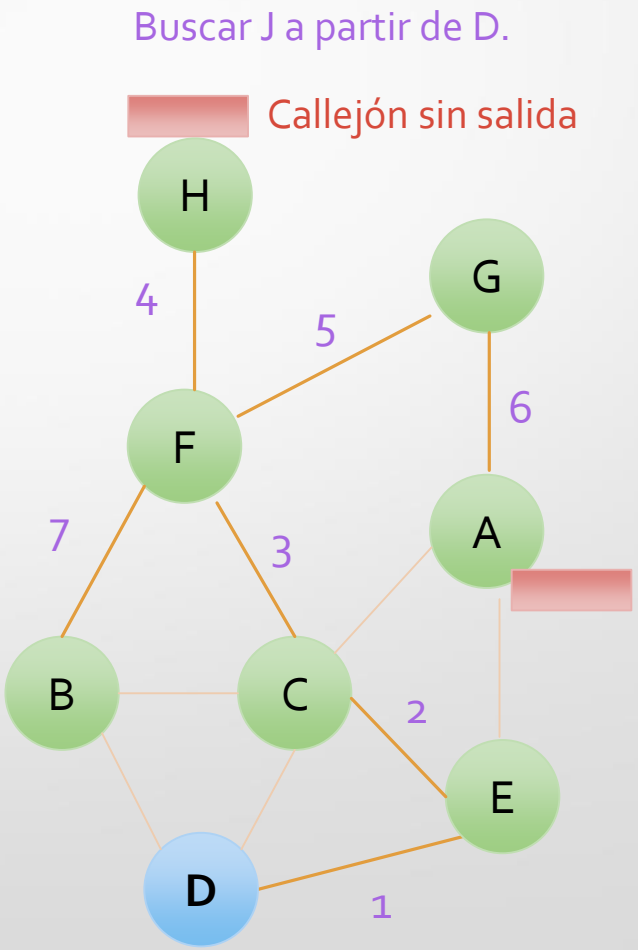



# Búsquedas en grafos.

- Hay un nodo inicial y una clave a buscar.
- Puede devolver el dato con dicha clave, o un simple *falso* o *verdadero* indicando si existe o no.
- Búsqueda en profundidad.
  - Depth-first search.
  - Puede implementarse mediante recursión o con pilas (iterativo).
- Búsqueda en anchura.
  - Breadth-first search.
  - Puede implementarse mediante colas (iterativo).

# Búsqueda en profundidad

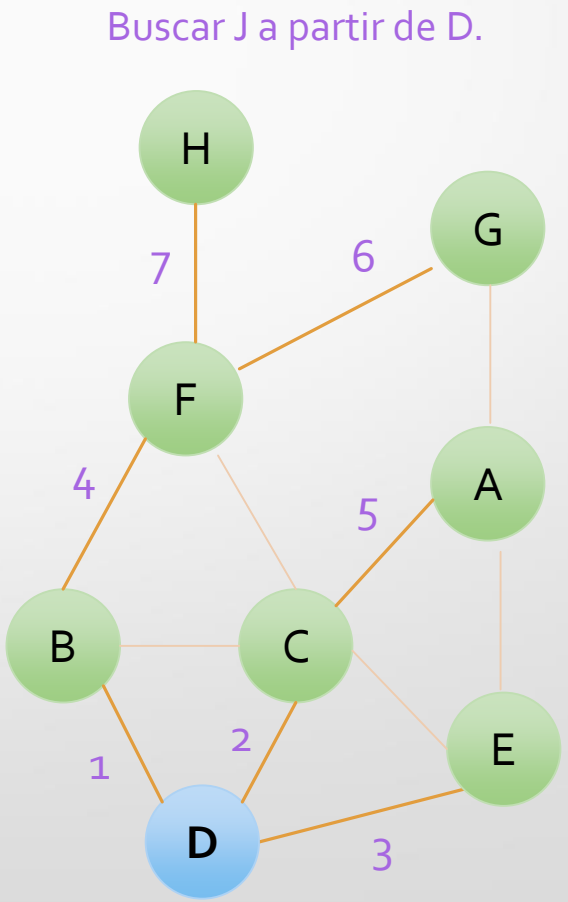
1. Comenzando del nodo raíz, la búsqueda en profundidad se mueve al primer vecino no visitado del actual mientras no encuentre la clave y no llegue a un callejón sin salida: ya no vecinos sin visitar.
2. Cuando llega a un callejón sin salida, efectúa un retroceso: regresa al penúltimo nodo visitado y procesa el siguiente vecino no visitado.
3. Al usar una pila de nodos por visitar, el último nodo añadido será el primero en procesarse.



# Búsqueda en anchura

1. Comenzando del nodo raíz, la búsqueda en anchura procesa todos los vecinos (distancia = 1), luego los vecinos de cada vecino (distancia = 2), y así sucesivamente.
2. Al usar una cola de nodos por visitar, los primeros nodos añadidos son los primeros en ser procesados.

- Ambas tipos de búsqueda necesitan una lista de nodos ya visitados, además de la cola/pila de nodos por visitar.



# Ejercicio

- Implemente un método que determine si una clave es accesible a partir de otra clave de un grafo no ponderado, no dirigido y no necesariamente conexo.
  - Utilizando búsqueda en profundidad y en anchura.
  - Utilizando claves enteras y alfanuméricas.
- Represente el grafo mediante una matriz de adyacencia.
- ¿Cómo son las estructuras de datos …
  - De entrada (contenido del grafo).
  - De procesamiento (nodos visitados, nodos por visitar).
- ¿Qué tanto aceleran la búsqueda las claves enteras?