Algoritmos y Estructuras de Datos II

Algoritmos voraces

Clase de hoy

1. Organización de la materia

2. Algoritmos voraces
   - Forma general
   - Problema de la moneda
   - Problema de la mochila

Organización de la materia

- cómo vs. qué
- 3 partes
  1. análisis de algoritmos
  2. tipos de datos
  3. técnicas de resolución de problemas
     - divide y vencerás
     - algoritmos voraces
     - backtracking
     - programación dinámica
     - recorrida de grafos

Algoritmos Voraces (o Glotones, Golosos) (Greedy)

- Es la técnica más sencilla de resolución de problemas.
- Normalmente se trata de algoritmos que resuelven problemas de **optimización**, es decir, tenemos un problema que queremos resolver de manera **óptima**:
  - el camino más corto que une dos ciudades,
  - el valor máximo alcanzable entre ciertos objetos,
  - el costo mínimo para proveer un cierto servicio,
  - el menor número de billetes para pagar un cierto importe,
  - el menor tiempo necesario para realizar un trabajo, etc.
- Los algoritmos voraces intentan construir la solución óptima buscada **paso a paso**,
- **eligiendo** en cada paso
- la **componente** de la solución
- que **parece** más apropiada.

Características

- Nunca revisan una **elección** ya realizada,
- confían en haber elegido bien las componentes anteriores.
- Por ello, lamentablemente, no todos los problemas admiten solución voraz,
- pero varios problemas interesantes sí admiten solución voraz,
- y entonces, dichas soluciones resultan muy eficientes.

Problemas con solución voraz

- Problema de la moneda (casos especial).
- Problema de la mochila (caso especial).
- Problema del camino de costo mínimo en un grafo.
- Problema del árbol generador de costo mínimo en un grafo.
- Muchos otros problemas menos conocidos.

Ingredientes comunes de los algoritmos voraces

- se tiene un problema a resolver de manera **óptima**,
- un conjunto de **candidatos** a integrar la solución,
- los candidatos se van clasificando en 3: los aún no considerados, los **incorporados** a la solución parcial, y los **descartados**,
- tenemos una manera de saber si los candidatos ya incorporados completan una **solución** del problema (sin preocuparse por si la misma es o no óptima),
- una función que comprueba si un candidato o un conjunto de candidatos es **factible** de formar parte de la solución.
- finalmente, otra función que **selecciona** de entre los candidatos aún no considerados, el más promisorio.

Receta general de los algoritmos voraces

- Inicialmente ningún candidato ha sido considerado, es decir, ni incorporado ni descartado.
- En cada paso se utiliza la función de **selección** para elegir cuál candidato considerar.
- Se chequea que el candidato considerado sea **factible** para incorporarlo a la solución y se lo agrega o no.
- Se repiten los pasos anteriores hasta que la colección de candidatos elegidos sea una solución.

Esquema general de algoritmo voraz

fun voraz(C : Set **of** “Candidato”) **ret** S : “Solución a construir”
    S:= “solución vacía”
    do S “no es solución” → c := “seleccionar” de C
                                elim(C,c)
                                if “agregar c a S es factible” →
                                        “agregar c a S”
                                fi
    od
end fun

Lo más importante es el criterio de selección.

Problema de la moneda

- Tenemos una cantidad infinita de monedas de cada una de las siguientes denominaciones:
  - 1 peso,
  - 50 centavos,
  - 25 centavos,
  - 10 centavos,
  - 5 centavos
  - y 1 centavo.
- Se desea pagar un cierto monto de manera exacta.
- Se debe determinar la manera de pagar dicho importe exacto con la menor cantidad de monedas posible.

Solución al problema de la moneda

- Seleccionar una moneda de la mayor denominación posible que no exceda el monto a pagar,
- utilizar exactamente el mismo algoritmo para el importe remanente.

Ya tenemos definido el **criterio de selección**.

Antes de implementar la solución, debemos pensar con qué **estructuras de datos** vamos a representar los elementos que intervienen en el problema. En este caso sencillo utilizamos naturales para cada denominación de moneda, un natural para el monto a pagar, y un natural para la solución que indica la cantidad de monedas necesarias para pagar.

Algoritmo voraz: versión 1

```
fun cambio(m: Nat, C: Set of Nat) ret S : Nat
    var c, resto: Nat
    var C_aux : Set of Nat
    S:= 0
    C_aux:= set_copy(C)
    resto:= m
    do resto > 0 →
        c := “elemento máximo de C_aux”
        elim(C_aux,c)
        S := S + resto ’div’ c
        resto := resto ’mod’ c
    od
    set_destroy(C_aux)
end fun
```

Algoritmo voraz: versión 2

```
fun cambio(m: Nat,C: Set of Nat) ret S : Nat
    var c, resto: Nat
    var C_aux : Set of Nat
    S:= 0
    C_aux:= set_copy(C)
    resto:= m
    do (not is_empty_set(C_aux)) →
                        c := "elemento máximo de C_aux" {selecciono}
                        if c > resto {chequeo factibilidad}
                          then elim(C_aux,c)
                          else resto := resto - c
                                S := S + 1 {agrego a la solución}
                        fi
    od
    set_destroy(C_aux)
end fun
```

Algoritmo voraz: devolviendo valor de cada moneda

```
fun cambio(m: Nat,C: Set of Nat) ret S : List of Nat
    var c, resto: Nat
    var C_aux : Set of Nat
    S:= empty_list()
    C_aux:= set_copy(C)
    resto:= m
    do (not is_empty_set(C_aux)) →
                        c := "elemento máximo de C_aux" {selecciono}
                        if c > resto {chequeo factibilidad}
                          then elim(C_aux,c)
                          else resto := resto - c
                                addl(S,c) {agrego a la solución}
                        fi
    od
    set_destroy(C_aux)
end fun
```

Sobre el algoritmo

- La versión 1 tiene el problema de que para algunos conjuntos de denominaciones puede que el ciclo no termine nunca o bien que la ejecución falle al querer obtener el máximo de un conjunto vacío.
- En ambos casos resta por implementar la función que elige el máximo de un conjunto.
- El algoritmo no siempre obtiene la solución óptima, depende del conjunto de denominaciones.
- Para un conjunto razonable, funciona.
- Para denominaciones habituales: 100, 50, 25, 10, 5 y 1 funciona.

Conjunto de denominaciones para el que no funciona

- Sean 4, 3 y 1 las denominaciones y sea 6 el monto a pagar.
- El algoritmo voraz intenta pagar con una moneda de denominación 4, queda un saldo de 2 que solamente puede pagarse con 2 monedas de 1, en total, 3 monedas.
- Pero hay una solución mejor: dos monedas de 3.
- De todas formas, el algoritmo anda bien para todas las denominaciones de uso habitual.

Problema de la mochila

- Tenemos una mochila de capacidad W.
- Tenemos n objetos de valor $v_1, v_2, \ldots, v_n$ y peso $w_1, w_2, \ldots, w_n$.
- Se quiere encontrar la mejor selección de objetos para llevar en la mochila.
- Por mejor selección se entiende aquélla que totaliza el mayor valor posible sin que su peso exceda la capacidad W de la mochila.
- Para que el problema no sea trivial, asumimos que la suma de los pesos de los n objetos excede la capacidad de la mochila, obligándonos entonces a seleccionar cuáles cargar en ella.

Criterio de selección

¿Cómo conviene seleccionar un objeto para cargar en la mochila?

- El más valioso de todos.
- El menos pesado de todos.
- Una combinación de los dos.

Análisis del primer criterio de selección

El más valioso primero

- Razonabilidad: el objetivo es cargar la mochila con el mayor valor posible, escogemos los objetos más valiosos.
- Falla: puede que al elegir un objeto valioso dejemos de lado otro apenas menos valioso pero mucho más liviano.
- Ejemplo: Mochila de capacidad 10, objetos de valor 12, 11 y 9, y peso 7, 5 y 5.
- De elegir primero el de mayor valor (12) ocuparíamos 7 de los 10 kg de la mochila, no quedando lugar para otro objeto.
- En cambio, de elegir el de valor 11, ocuparíamos solamente 5 kg quedando 5 kg para el de valor 9, totalizando un valor de 20.

Análisis del segundo criterio de selección

El menos pesado primero

- Razonabilidad: hay que procurar aprovechar la capacidad de la mochila, escogemos los objetos más livianos.
- Falla: puede que al elegir un objeto liviano dejemos de lado otro apenas más pesado pero mucho más valioso.
- Ejemplo: Mochila de capacidad 13, objetos de valor 12, 11 y 7, y peso 6, 6 y 5.
- De elegir primero el de menor peso (5) obtendríamos su valor (7) más, en el mejor de los casos, 12, totalizando 12+7=19.
- En cambio, de elegir los dos de peso 6, no se excede la capacidad de la mochila y se totaliza un valor de 23.

Análisis del tercer criterio de selección

Combinando ambos criterios

- Debemos asegurarnos de que cada kg utilizado de la mochila sea aprovechado de la mejor manera posible: que cada kg colocado en la mochila valga lo más posible.
- Criterio: elegir el de mayor valor relativo (cociente entre el valor y el peso): dicho cociente expresa el valor promedio de cada kg de ese objeto.
- Falla: puede que al elegir un objeto dejemos de lado otro de peor cociente, pero que aprovecha mejor la capacidad.
- Ejemplo: Mochila de capacidad 10, objetos de valor 12, 11 y 8, y peso 6, 5 y 4.
- El critero elige al que pesa 5, ya que cada kg de ese objeto vale más de 2. Pero convenía elegir los otros dos.

Problema de la mochila
Versión simplificada

- El problema de la mochila no admite solución voraz.
- Se simplifica permitiendo **fraccionar** objetos.
- Ahora sí el tercer criterio funciona.
- (En el ejemplo anterior, elegimos primero el que vale 11 y luego 5/6 del que vale 12 obteniendo como valor total 11 + 10 = 21).

Ya tenemos definido criterio de selección. Antes de implementar el algoritmo definimos un tipo de datos para representar los objetos de la mochila.

Estructura de datos necesaria

```
type Objeto = tuple
                  id : Nat
                  value: Float
                  weight: Float
              end tuple
```

Resolveremos el problema asumiendo que los datos vienen dados por un conjunto C : Set **of** Objeto.
La solución la podemos representar por una lista donde cada elemento es un par conteniendo el objeto y la fracción que agrego de él:

```
type Obj_Mochila = tuple
                       obj : Objeto
                       fract : Float
                   end tuple
```

Algoritmo voraz

```
fun mochila(W: Float, C: Set of Objeto) ret L : List of Obj_Mochila
    var o_m : Obj_Mochila     var resto : Float
    var C_aux : Set of Objeto
    S:= empty_list()
    C_aux:= set_copy(C)
    resto:= W
    do (resto > 0) →
        o_m.obj := select_obj(C_aux)
        if o_m.obj.weight <= resto
          then o_m.fract := 1.0
                resto := resto - o_m.obj.weight
          else o_m.fract := resto/o_m.obj.weight
               resto := 0
        fi
        addl(S,o_m)
        elim(C_aux,o_m.obj)
    od
    set_destroy(C_aux)
end fun
```

Algoritmo voraz

```
fun select_obj(C: Set of Objeto) ret r : Objeto
    var C_aux : Set of Objeto
    var o : Objeto
    var m : Float
    m := -∞
    C_aux := set_copy(C)
    do (not is_empty_set(C_aux)) →
        o := get(C_aux)
        if (o.value/o.weight > m)
          then m := o.value/o.weight
               r := o
        fi
        elim(C_aux,o)
    od
    set_destroy(C_aux)
end fun
```

Sobre este algoritmo

- funciona siempre que esté permitido fraccionar objetos.
- la función select_obj implementa el criterio de selección, eligiendo del conjunto de objetos, aquel que tiene la relación valor/peso máxima.
- La solución al problema puede implementarse también utilizando arreglos.