Algoritmos y Estructuras de Datos II

Backtracking

Clase de hoy

1. Backtracking
   - Forma general de algoritmos voraces
   - ¿Y cuando no hay un buen criterio de selección?
   - Problema de la moneda
   - Problema de la mochila
   - Camino de costo mínimo entre todo par de vértices
2. Conclusiones
3. Backtracking, grafo implícito

Forma general de algoritmos voraces

```
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
```

- Ser solución y ser factible no tienen en cuenta optimalidad.
- Optimalidad depende totalmente del criterio de selección.

¿Y cuando no hay un buen criterio de selección?

- A veces no hay un criterio de selección que garantice optimalidad.
- Por ejemplo:
  - Problema de la moneda para conjuntos de denominaciones arbitrarios.
  - Problema de la mochila para objetos no fraccionables.
- En este caso, si se elige un fragmento de solución puede ser necesario "volver hacia atrás" (**backtrack**) sobre esa elección e intentar otro fragmento.
- En la práctica, estamos hablando de considerar todas las selecciones posibles e intentar cada una de ellas para saber cuál de ellas conduce a la solución óptima. (backtracking = fuerza bruta)

A diferencia de la técnica voraz

- Siempre que haya solución, backtracking la encuentra.
- En general son algoritmos ineficientes (aunque pueda que no se conozcan mejores alternativas).
- No hay buen criterio de selección: se utiliza fuerza bruta.
- A veces se puede ser un poco menos brutal...

Problema de la moneda

- Sean $d_1, d_2, \ldots, d_n$ las denominaciones de las monedas (todas mayores que 0),
- no se asume que estén ordenadas,
- se dispone de una cantidad infinita de monedas de cada denominación,
- se desea pagar un monto k de manera exacta,
- utilizando el **menor número de monedas posibles**.
- Vimos que el algoritmo voraz puede no funcionar para ciertos conjuntos de denominaciones.
- Daremos un algoritmo recursivo consistente en considerar todas las combinaciones de monedas posibles.

Problema de la moneda

- Dado un conjunto con las n denominaciones, y un monto k a pagar,
- recursivamente iremos probando con cada denominación pagar el monto,
- si una denominación es factible (no se pasa del monto), pruebo usarla o no.
- Para ello calculo el resultado en cada caso, y luego obtengo el mínimo.
- El algoritmo devolverá la menor cantidad de monedas necesarias.

Problema de la moneda usando backtracking

```
fun cambio(j : Nat, C: Set of Nat) ret S : Nat
    var c, Nat
    var C_aux : Set of Nat
    if j = 0 then S := 0
    else if is_empty(C) then S := ∞
    else C_aux:= set_copy(C)
         c := get(C)
         elim(C_aux,c)
         if (c ≤ j) then S := min(1+cambio(j-c,C),cambio(j,C_aux))
                    else S := cambio(j,C_aux)
         fi
         set_destroy(C_aux)
    fi
end fun
```

La solución la obtenemos llamando a cambio(k,C), donde C contiene los valores $d_1, d_2, ..., d_n$.

Problema de la moneda

- El primer caso del if corresponde a cuando ya llegamos a pagar el total, en cuyo caso no necesito más monedas.
- El segundo caso es cuando he probado descartar tantas denominaciones que no puedo pagar el monto, por lo cual devolvemos infinito, ya que ese *intento* falló.
- En el tercer caso observo si la denominación es factible para pagar el monto j, en cuyo caso intento utilizarla o no, quedándome con el resultado mínimo.

Otra implementación

Observemos que no nos hace falta llevar un conjunto de denominaciones. Podría simplemente tomar un natural i que me indique que estoy considerando las denominaciones $d_1, d_2, \ldots, d_i$, y el monto j que debo pagar:

```
fun cambio(d:array[1..n] of nat, i,j: nat) ret r: nat
    if j=0 then r:= 0
    else if i = 0 then r:= ∞
    else if d[i] > j then r:= cambio(d,i-1,j)
    else r:= min(cambio(d,i-1,j),1+cambio(d,i,j-d[i]))
    fi
end fun
```

La solución la obtenemos al llamar a cambio(d,n,k), donde d es un arreglo conteniendo el varlor de las denominaciones en cada posición.

Definición recursiva con otra notación

Como lo único interesante de estas soluciones con backtracking es la función recursiva, podemos definirla matemáticamente:

cambio$(i, j) =$ *"menor número de monedas necesarias para pagar exactamente el monto* j *con denominaciones* $d_1, d_2, \ldots, d_i$*."*

$$cambio(i,j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = 0 \\ cambio(i-1,j) & d_i > j > 0 \wedge i > 0 \\ \min(cambio(i-1,j), 1 + cambio(i, j-d_i)) & j \geq d_i > 0 \wedge i > 0 \end{cases}$$

El resultado al problema lo obtengo llamando a *cambio*(n, k).

Problema de la mochila

- Tenemos una mochila de capacidad W.
- Tenemos n objetos **no fraccionables** 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.

Simplificación y generalización

- Simplificamos el problema:
  - sólo nos interesa por ahora hallar el mayor valor posible sin exceder la capacidad de la mochila,
  - no nos interesa saber cuáles son los objetos que alcanzan ese máximo.
- De manera similar al problema de la moneda, definimos una función recursiva $m(i, j)$, representando con el parámetro i que consideramos los objetos $1, 2, ..., i$ y con j la capacidad restante de la mochila.

Problema de la mochila

Definimos $m(i, j) =$ "mayor valor alcanzable sin exceder la capacidad j con objetos $1, 2, \ldots, i$."

$$m(i,j) = \begin{cases} 0 & j = 0 \\ 0 & j > 0 \wedge i = 0 \\ m(i-1,j) & w_i > j > 0 \wedge i > 0 \\ \max(m(i-1,j), v_i + m(i-1, j-w_i)) & j \geq w_i > 0 \wedge i > 0 \end{cases}$$

El resultado al problema lo obtenemos llamando a $m(n, W)$.

Problema del camino de costo mínimo

Entre todo par de vértices

- Tenemos un grafo dirigido $G = (V, A)$,
- con costos no negativos en las aristas,
- se quiere encontrar, para cada par de vértices, el camino de menor costo que los une.
- Se asume $V = \{1, \ldots, n\}$

Simplificación y generalización

- Simplificamos el problema:
  - sólo nos interesa por ahora hallar el costo de cada uno de los caminos de costo mínimo.
  - no nos interesa saber cuáles son los caminos que alcanzan ese mínimo.
- Generalizamos el problema:
  - Sean $1 \leq i, j \leq n$ y $0 \leq k \leq n$,
  - definimos $c(k, i, j) =$ "menor costo posible para caminos de i a j cuyos vértices intermedios se encuentran en el conjunto $\{1,\ldots,k\}$."
  - La solución del problema original se obtiene calculando $c(n, i, j)$ para el par i (origen) y j (destino) que se desea.

Definición recursiva de *camino*

$$c(k,i,j) = \begin{cases} L[i,j] & k = 0 \\ \min(c(k-1,i,j), c(k-1,i,k) + c(k-1,k,j)) & k \geq 1 \end{cases}$$

donde $L[i,j]$ es el costo de la arista que va de i a j, o infinito si no hay tal arista.

Conclusiones

En cada problema el algoritmo calcula cada resultado de acuerdo a la decisión de agregar un candidato a la solución. Y se queda con el mejor.

- En el problema de la moneda decido si utilizo o no utilizo la denominación i-ésima.
- En el problema de la mochila decido si utilizo o no el objeto i-ésimo.
- En el problema del camino de costo mínimo decido si paso o no por el vértice k.

Conclusiones

- Hemos visto soluciones a tres problemas.
- En general, muy ineficiente.
- Por ejemplo, para el problema de la moneda, si queremos pagar el monto 90 con denominaciones 1, 5 y 10,
  - cambio(3,90) llama a cambio(2,90) y cambio(3,80),
  - cambio(2,90) llama a cambio(1,90) y cambio(2,85),
  - cambio(2,85) llama a cambio(1,85) y **cambio(2,80)**,
  - cambio(3,80) llama a **cambio(2,80)** y cambio(3,70).
- Se ve que cambio(2,80) se calcula 2 veces.
- y muchos otros llamados se repiten, incluso varias veces.
- Los algoritmos son exponenciales.

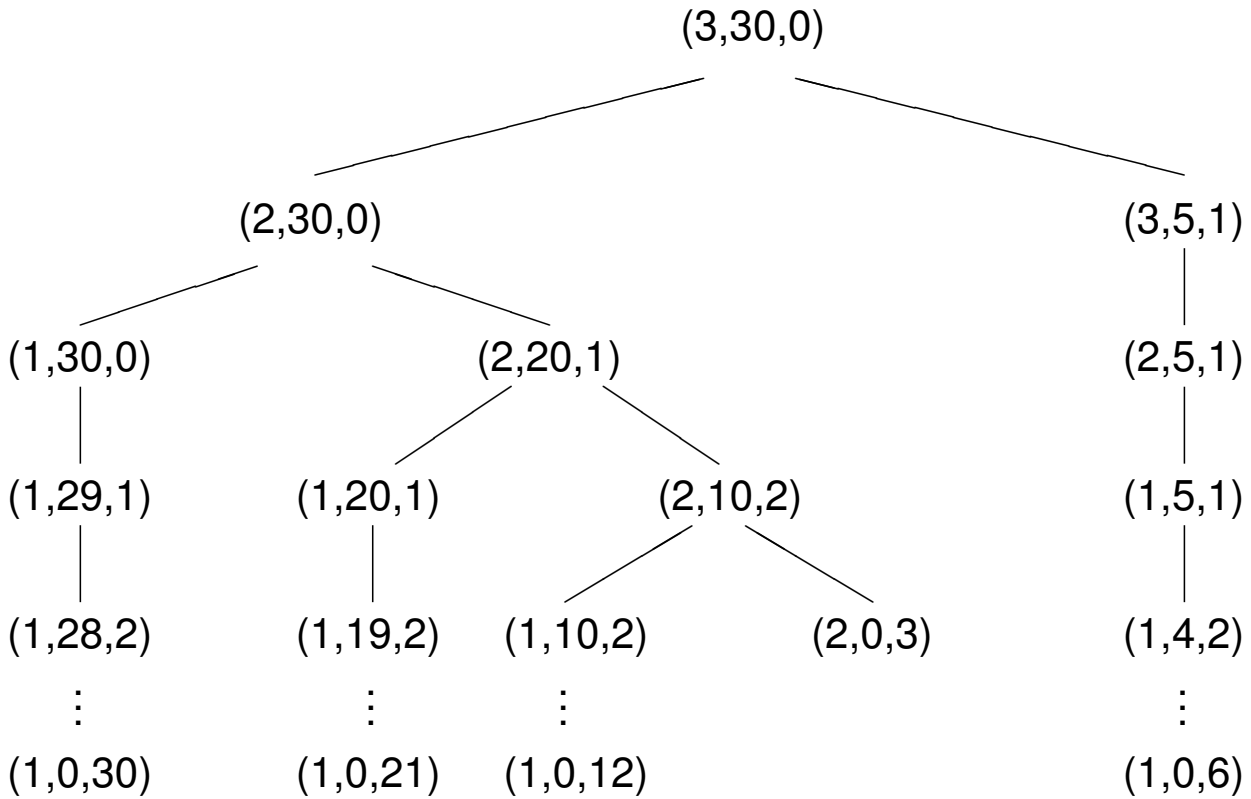
Problema de la moneda

$$cambio(i,j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = 0 \\ cambio(i-1,j) & d_i > j > 0 \wedge i > 0 \\ \min(cambio(i-1,j), 1 + cambio(i, j-d_i)) & j \geq d_i > 0 \wedge i > 0 \end{cases}$$

Podemos dibujar un árbol indicando en cada vértice la llamada a la función $cambio(i,j)$ indicando el resultado obtenido hasta ese momento. El vértice (i,j,x) indica que llamo a $cambio(i,j)$ y el resultado hasta ese momento es x.

Grafo implícito

Ejemplo $d_1 = 1$, $d_2 = 10$, $d_3 = 25$ y $k = 30$



Grafo implícito
Definición general

- Desde el vértice (i, j, x), si $i, j > 0$ y $d_i < j$ existe una única arista al vértice $(i - 1, j, x)$.
- En cambio si $j \leq d_i$ existen dos aristas:
  - una a $(i - 1, j, x)$
  - y otra a $(i, j - d_i, x + 1)$.
- la raíz es el vértice $(n, k, 0)$.