Comparación de técnicas de diseño

Clase 19

IIC 2133 - Sección 1

Prof. Diego Arroyuelo

Sumario

Nuestra cuarta estrategia de diseño

Nuestra cuarta estrategia de diseño

Programación dinámica

Nuestra cuarta estrategia de diseño

Programación dinámica

- Generalmente usada en problemas de optimización

Nuestra cuarta estrategia de diseño

Programación dinámica

- Generalmente usada en problemas de optimización
- Se basa en la existencia de **subproblemas** que permiten resolver el problema original

Nuestra cuarta estrategia de diseño

Programación dinámica

- Generalmente usada en problemas de optimización
- Se basa en la existencia de **subproblemas** que permiten resolver el problema original
- Además, los subproblemas se **solapan**, i.e. comparten **sub-subproblemas**

Nuestra cuarta estrategia de diseño

Programación dinámica

- Generalmente usada en problemas de optimización
- Se basa en la existencia de **subproblemas** que permiten resolver el problema original
- Además, los subproblemas se **solapan**, i.e. comparten **sub-subproblemas**

La diferencia con **dividir para conquistar** es que en esta última los subproblemas son disjuntos

Nuestra cuarta estrategia de diseño

Programación dinámica

- Generalmente usada en problemas de optimización
- Se basa en la existencia de **subproblemas** que permiten resolver el problema original
- Además, los subproblemas se **solapan**, i.e. comparten **sub-subproblemas**

La diferencia con **dividir para conquistar** es que en esta última los subproblemas son disjuntos

La clave de programación dinámica es
recordar las soluciones a los subproblemas

Problema de dar vuelto

Problema de dar vuelto

Consideremos ahora el problema de dar S pesos de vuelto usando el menor número posible de monedas

Problema de dar vuelto

Consideremos ahora el problema de dar S pesos de vuelto usando el menor número posible de monedas

- Suponemos que los valores de las monedas, ordenados de mayor a menor, son $\{v_1, v_2 \ldots, v_n\}$

Problema de dar vuelto

Consideremos ahora el problema de dar S pesos de vuelto usando el menor número posible de monedas

- Suponemos que los valores de las monedas, ordenados de mayor a menor, son $\{v_1, v_2 \ldots, v_n\}$
- Tenemos una cantidad ilimitada de monedas de cada valor

Problema de dar vuelto

Consideremos ahora el problema de dar S pesos de vuelto usando el menor número posible de monedas

- Suponemos que los valores de las monedas, ordenados de mayor a menor, son $\{v_1, v_2 \ldots, v_n\}$
- Tenemos una cantidad ilimitada de monedas de cada valor

Posible estrategia codiciosa:

Problema de dar vuelto

Consideremos ahora el problema de dar S pesos de vuelto usando el menor número posible de monedas

- Suponemos que los valores de las monedas, ordenados de mayor a menor, son $\{v_1, v_2 \ldots, v_n\}$
- Tenemos una cantidad ilimitada de monedas de cada valor

Posible estrategia codiciosa:

Asignar tantas monedas *grandes* como sea posible, antes de avanzar a la siguiente moneda *grande*

Problema de dar vuelto

Consideremos ahora el problema de dar S pesos de vuelto usando el menor número posible de monedas

- Suponemos que los valores de las monedas, ordenados de mayor a menor, son $\{v_1, v_2 \ldots, v_n\}$
- Tenemos una cantidad ilimitada de monedas de cada valor

Posible estrategia codiciosa:

Asignar tantas monedas *grandes* como sea posible, antes de avanzar a la siguiente moneda *grande*

Ejemplo

Si $\{v_1, v_2, v_3, v_4\} = \{10, 5, 2, 1\}$, la estrategia codiciosa **siempre** produce el menor número de monedas para un vuelto S cualquiera

Problema de dar vuelto

Ejemplo

Sin embargo, la estrategia no funciona para un conjunto de valores cualquiera.

Problema de dar vuelto

Ejemplo

Sin embargo, la estrategia no funciona para un conjunto de valores cualquiera. Si $\{v_1, v_2, v_3\} = \{6, 4, 1\}$ y $S = 8$, entonces la estrategia produce

$$8 = 6 + 1 + 1$$

pero el óptimo es

$$8 = 4 + 4$$

Problema de dar vuelto

Ejemplo

Sin embargo, la estrategia no funciona para un conjunto de valores cualquiera. Si $\{v_1, v_2, v_3\} = \{6, 4, 1\}$ y $S = 8$, entonces la estrategia produce

$$8 = 6 + 1 + 1$$

pero el óptimo es

$$8 = 4 + 4$$

Lo atacamos con programación dinámica

Problema de dar vuelto

Dado un conjunto de valores ordenados $\{v_1, \ldots, v_n\}$, definimos $z(S, n)$ como el problema de encontrar el menor número de monedas para totalizar S

Problema de dar vuelto

Dado un conjunto de valores ordenados $\{v_1, \ldots, v_n\}$, definimos $z(S, n)$ como el problema de encontrar el menor número de monedas para totalizar S

Ejercicio

Proponga una recurrencia para resolver el problema $z(S, n)$ y plantee un algoritmo a partir de ella.

Problema de dar vuelto

Ejercicio

Sea $Z(S, n)$ la solución óptima al problema $z(S, n)$. Para buscar intuición, notamos que hay dos opciones respecto a las monedas de valor v_n

Problema de dar vuelto

Ejercicio

Sea $Z(S, n)$ la solución óptima al problema $z(S, n)$. Para buscar intuición, notamos que hay dos opciones respecto a las monedas de valor v_n

- Si se incluye una moneda de valor v_n,

$$Z(S, n) = Z(S - v_n, n) + 1$$

Problema de dar vuelto

Ejercicio

Sea $Z(S, n)$ la solución óptima al problema $z(S, n)$. Para buscar intuición, notamos que hay dos opciones respecto a las monedas de valor v_n

- Si se incluye una moneda de valor v_n,

$$Z(S, n) = Z(S - v_n, n) + 1$$

- Si no se usan monedas de valor v_n,

$$Z(S, n) = Z(S, n - 1)$$

Problema de dar vuelto

Ejercicio

Luego, generalizamos esta idea

Problema de dar vuelto

Ejercicio

Luego, generalizamos esta idea

- Para las monedas de de valor v_n,

$$Z(S, n) = \min\{Z(S - v_n, n) + 1,\ Z(S, n - 1)\}$$

Problema de dar vuelto

Ejercicio

Luego, generalizamos esta idea

- Para las monedas de de valor v_n,

$$Z(S, n) = \min\{Z(S - v_n, n) + 1,\ Z(S, n - 1)\}$$

- Luego, generalizamos para el subconjunto de los primeros k valores de monedas

$$Z(T, k) = \min\{Z(T - v_k, n) + 1,\ Z(T, k - 1)\}$$

donde $Z(T, 0) = +\infty$ si $T > 0$, y $Z(0, k) = 0$

Problema de dar vuelto

Ejercicio

Con esto, podemos plantear el siguiente algoritmo iterativo

Change(S):

    for $T = 1, \ldots, S$ **:**

        $Z[T][0] \leftarrow +\infty$

    for $k = 0, \ldots, n$ **:**

        $Z[0][k] \leftarrow 0$

    for $k = 1, \ldots, n$ **:**

        for $T = 1, \ldots, S$ **:**

            $Z[T][k] \leftarrow Z[T][k-1]$

            if $T - v_k \geq 0$ **:**

                $Z[T][k] \leftarrow \min\{Z[T][k],\ Z[T - v_k, k]\}$

Objetivos de la clase

Objetivos de la clase

- Comparar las técnicas estudiadas

Objetivos de la clase

- Comparar las técnicas estudiadas
- Aplicar las técnicas para resolver problemas

Sumario

Panorama de técnicas

En este punto ya conocemos las 4 estrategias siguientes

- Dividir para conquistar
- Backtracking
- Algoritmos codiciosos
- Programación dinámica

¿Cuándo preferimos una sobre otra en la práctica?

Panorama de técnicas

No todas las estrategias resuelven los mismos problemas

- ¿Problemas con subproblemas del mismo tipo?
- ¿Subestructura óptima?
- ¿Problema de satisfacción de restricciones?
- ¿Problema con estrategia codiciosa?
- ¿Subproblemas se repiten?

La práctica nos sugiere qué técnicas son relevantes ante un problema

Panorama de técnicas

Caso especial: problemas de optimización

- En general, se pueden resolver con más de una técnica

¿Qué criterios permiten escoger?

Optimización y las técnicas

Backtracking

Optimización y las técnicas

Backtracking

- Requiere computar **todas** las soluciones factibles

Optimización y las técnicas

Backtracking

- Requiere computar **todas** las soluciones factibles
- Eso puede ser útil en ciertas ocasiones
- Muy costoso en tiempo

Optimización y las técnicas

Backtracking

- Requiere computar **todas** las soluciones factibles
- Eso puede ser útil en ciertas ocasiones
- Muy costoso en tiempo

Codiciosos

Optimización y las técnicas

Backtracking

- Requiere computar **todas** las soluciones factibles
- Eso puede ser útil en ciertas ocasiones
- Muy costoso en tiempo

Codiciosos

- Solo si es que existe estrategia probada

Optimización y las técnicas

Backtracking

- Requiere computar **todas** las soluciones factibles
- Eso puede ser útil en ciertas ocasiones
- Muy costoso en tiempo

Codiciosos

- Solo si es que existe estrategia probada
- MUY eficientes en tiempo y memoria

Optimización y las técnicas

Backtracking

- Requiere computar **todas** las soluciones factibles
- Eso puede ser útil en ciertas ocasiones
- Muy costoso en tiempo

Codiciosos

- Solo si es que existe estrategia probada
- MUY eficientes en tiempo y memoria

Programación dinámica

Optimización y las técnicas

Backtracking

- Requiere computar **todas** las soluciones factibles
- Eso puede ser útil en ciertas ocasiones
- Muy costoso en tiempo

Codiciosos

- Solo si es que existe estrategia probada
- MUY eficientes en tiempo y memoria

Programación dinámica

- Solo si hay subestructura óptima

Optimización y las técnicas

Backtracking

- Requiere computar **todas** las soluciones factibles
- Eso puede ser útil en ciertas ocasiones
- Muy costoso en tiempo

Codiciosos

- Solo si es que existe estrategia probada
- MUY eficientes en tiempo y memoria

Programación dinámica

- Solo si hay subestructura óptima
- Puede ser eficiente en tiempo y memoria

Sumario

Coloración de grafos

Coloración de grafos

Definición

Una k-**coloración** de un grafo o árbol $G = (V, E)$ es una función $f : V \to \{1, \ldots, k\}$ que asigna un color a cada vértice de G de forma que vértices conectados por una arista no tienen el mismo color.

Coloración de grafos

Definición

Una ***k*-coloración** de un grafo o árbol $G = (V, E)$ es una función $f : V \to \{1, \ldots, k\}$ que asigna un color a cada vértice de G de forma que vértices conectados por una arista no tienen el mismo color. Decimos que G es ***k*-coloreable** si existe una k-coloración para sus vértices.

Coloración de grafos

Definición

Una ***k*-coloración** de un grafo o árbol $G = (V, E)$ es una función $f : V \to \{1, \dots, k\}$ que asigna un color a cada vértice de G de forma que vértices conectados por una arista no tienen el mismo color. Decimos que G es ***k*-coloreable** si existe una k-coloración para sus vértices.

Problema:	$\textsc{Col}$
Input:	Un grafo G y un natural $k \geq 1$
Output:	¿G es k-coloreable?

Coloración de grafos

Definición

Una ***k*-coloración** de un grafo o árbol $G = (V, E)$ es una función $f : V \to \{1, \ldots, k\}$ que asigna un color a cada vértice de G de forma que vértices conectados por una arista no tienen el mismo color. Decimos que G es ***k*-coloreable** si existe una k-coloración para sus vértices.

Problema:	$\textsc{Col}$
Input:	Un grafo G y un natural $k \geq 1$
Output:	¿G es k-coloreable?

¿Qué estrategias podemos emplear para dar una solución algorítmica?

Coloración de grafos

Problema:	COL
Input:	Un grafo G y un natural $k \geq 1$
Output:	¿G es k-coloreable?

COL involucra asignar colores bajo restricciones

Coloración de grafos

Problema:	COL
Input:	Un grafo G y un natural $k \geq 1$
Output:	¿G es k-coloreable?

COL involucra asignar colores bajo restricciones

- ¿Cómo se ve f que sea k-coloración?

Coloración de grafos

Problema:	COL
Input:	Un grafo G y un natural $k \geq 1$
Output:	¿G es k-coloreable?

COL involucra asignar colores bajo restricciones

- ¿Cómo se ve f que sea k-coloración?
- ¿Podemos comprobar fácilmente si f es k-coloración?

Coloración de grafos

Problema:	COL
Input:	Un grafo G y un natural $k \geq 1$
Output:	¿G es k-coloreable?

COL involucra asignar colores bajo restricciones

- ¿Cómo se ve f que sea k-coloración?
- ¿Podemos comprobar fácilmente si f es k-coloración?

Nos encontramos frente a un CSP

Coloración de grafos

Plan de diseño de solución con **Backtracking**

Coloración de grafos

Plan de diseño de solución con **Backtracking**

1. Suponemos conocido el listado de nodos (n nodos) y de aristas (m aristas)

Coloración de grafos

Plan de diseño de solución con **Backtracking**

1. Suponemos conocido el listado de nodos (n nodos) y de aristas (m aristas)
2. Escoger EDD para almacenar **asignación**: e.g. arreglo $A[0 \ldots n-1]$

Coloración de grafos

Plan de diseño de solución con **Backtracking**

1. Suponemos conocido el listado de nodos (n nodos) y de aristas (m aristas)
2. Escoger EDD para almacenar **asignación**: e.g. arreglo $A[0 \ldots n-1]$
3. Valores guardados se corresponden con los colores

Coloración de grafos

Plan de diseño de solución con **Backtracking**

1. Suponemos conocido el listado de nodos (n nodos) y de aristas (m aristas)
2. Escoger EDD para almacenar **asignación**: e.g. arreglo $A[0 \ldots n-1]$
3. Valores guardados se corresponden con los colores
4. Suponemos conocidos los vecinos $N(v)$ del nodo v

Coloración de grafos

Plan de diseño de solución con **Backtracking**

1. Suponemos conocido el listado de nodos (n nodos) y de aristas (m aristas)
2. Escoger EDD para almacenar **asignación**: e.g. arreglo $A[0 \ldots n-1]$
3. Valores guardados se corresponden con los colores
4. Suponemos conocidos los vecinos $N(v)$ del nodo v
5. Con $N(v)$ revisamos la **restricción de color**

Coloración de grafos

Plan de diseño de solución con **Backtracking**

1. Suponemos conocido el listado de nodos (n nodos) y de aristas (m aristas)
2. Escoger EDD para almacenar **asignación**: e.g. arreglo $A[0 \dots n-1]$
3. Valores guardados se corresponden con los colores
4. Suponemos conocidos los vecinos $N(v)$ del nodo v
5. Con $N(v)$ revisamos la **restricción de color**

Definimos un pseudocódigo para resolver el problema de decisión $\textsc{Col}$

Coloración de grafos

Suponemos que los nodos tienen etiquetas $V = \{0, \ldots, n-1\}$ y que el arreglo $A[0 \ldots n-1]$ comienza con valores `null`

Coloración de grafos

Suponemos que los nodos tienen etiquetas $V = \{0, \dots, n-1\}$ y que el arreglo $A[0 \dots n-1]$ comienza con valores `null`

input : arreglo $A[0 \dots n-1]$, número natural $k \geq 1$, índice $i \in \{0, \dots, n\}$

```
Col(A, k, i):
    if i = n :
        return True
    for j = 1, ..., k :
        if vecinos de i en N(i) tienen color distinto a j :
            A[i] ← j
            if Col(A, k, i + 1) :
                return True
        A[i] ← null
    return False
```

Coloración de grafos

Suponemos que los nodos tienen etiquetas $V = \{0, \dots, n-1\}$ y que el arreglo $A[0 \dots n-1]$ comienza con valores `null`

input : arreglo $A[0 \dots n-1]$, número natural $k \geq 1$, índice $i \in \{0, \dots, n\}$

```
Col(A, k, i):
1   if i = n :
2       return True
3   for j = 1, ..., k :
4       if vecinos de i en N(i) tienen color distinto a j :
5           A[i] ← j
6           if Col(A, k, i + 1) :
7               return True
8       A[i] ← null
9   return False
```

¿Cómo se implementaría la línea 4?

Coloración de grafos

A partir del pseudocódigo anterior, podemos obtener **todas** las coloraciones existentes

input : arreglo $A[0 \ldots n-1]$, número natural $k \geq 1$, índice $i \in \{0, \ldots, n\}$

```
Col(A, k, i):
    if i = n :
        return True
    for j = 1, ..., k :
        if vecinos de i en N(i) tienen color distinto a j :
            A[i] ← j
            if Col(A, k, i + 1) :
                return True
        A[i] ← null
    return False
```

¿Dónde realizamos modificaciones para obtener todas las soluciones?

Otro problema de asignación

Otro problema de asignación

Definición

Sea $A = \langle v_1, v_2, \ldots, v_n \rangle$ una secuencia de números naturales. Una **subsecuencia creciente** de A es una secuencia $B = \langle u_1, u_2, \ldots, u_m \rangle$ tal que $m \leq n$, cada elemento de B está en A y para todo $1 \leq i \leq m - 1$ se tiene que $u_i < u_{i+1}$. Considere el problema de encontrar la **subsecuencia creciente más larga** (SCML) para una secuencia.

Problema:	LIS
Input:	Una secuencia de naturales $A = \langle v_1, v_2, \ldots, v_n \rangle$
Output:	Subsecuencia creciente más larga $B = \langle u_1, u_2, \ldots, u_m \rangle$

Otro problema de asignación

Otro problema de asignación

Ejemplo

Para $A = \langle 3, 1, 5, 6\rangle$, hay dos SCML de largo 3 dadas por $B_1 = \langle 3, 5, 6\rangle$ y $B_2 = \langle 1, 5, 6\rangle$.

Problema:	LIS
Input:	Una secuencia de naturales $A = \langle v_1, v_2, \ldots, v_n\rangle$
Output:	Subsecuencia creciente más larga $B = \langle u_1, u_2, \ldots, u_m\rangle$

Otro problema de asignación

Ejemplo

Para $A = \langle 3, 1, 5, 6 \rangle$, hay dos SCML de largo 3 dadas por $B_1 = \langle 3, 5, 6 \rangle$ y $B_2 = \langle 1, 5, 6 \rangle$.

Problema:	LIS
Input:	Una secuencia de naturales $A = \langle v_1, v_2, \ldots, v_n \rangle$
Output:	Subsecuencia creciente más larga $B = \langle u_1, u_2, \ldots, u_m \rangle$

¿Qué estrategias podemos emplear para dar una solución algorítmica?

Otro problema de asignación

A diferencia del anterior, este es un problema de optimización

Otro problema de asignación

A diferencia del anterior, este es un problema de optimización

- También se puede ver como un CSP

Otro problema de asignación

A diferencia del anterior, este es un problema de optimización

- También se puede ver como un CSP
- ... con el ingrediente adicional de ser un problema que busca minimizar

Otro problema de asignación

A diferencia del anterior, este es un problema de optimización

- También se puede ver como un CSP
- ... con el ingrediente adicional de ser un problema que busca minimizar

¿Existe subestructura óptima de forma intuitiva?

Otro problema de asignación

A diferencia del anterior, este es un problema de optimización

- También se puede ver como un CSP
- ... con el ingrediente adicional de ser un problema que busca minimizar

¿Existe subestructura óptima de forma intuitiva?

Posibles abordajes

Otro problema de asignación

A diferencia del anterior, este es un problema de optimización

- También se puede ver como un CSP
- ... con el ingrediente adicional de ser un problema que busca minimizar

¿Existe subestructura óptima de forma intuitiva?

Posibles abordajes

- Backtracking:

Otro problema de asignación

A diferencia del anterior, este es un problema de optimización

- También se puede ver como un CSP
- ... con el ingrediente adicional de ser un problema que busca minimizar

¿Existe subestructura óptima de forma intuitiva?

Posibles abordajes

- Backtracking:
    - Intentar tomar todas las subsecuencias

Otro problema de asignación

A diferencia del anterior, este es un problema de optimización

- También se puede ver como un CSP
- ... con el ingrediente adicional de ser un problema que busca minimizar

¿Existe subestructura óptima de forma intuitiva?

Posibles abordajes

- Backtracking:
  - Intentar tomar todas las subsecuencias
  - Verificar que con cada una cumpla la definición de subsecuencia creciente

Otro problema de asignación

A diferencia del anterior, este es un problema de optimización

- También se puede ver como un CSP
- ... con el ingrediente adicional de ser un problema que busca minimizar

¿Existe subestructura óptima de forma intuitiva?

Posibles abordajes

- Backtracking:
  - Intentar tomar todas las subsecuencias
  - Verificar que con cada una cumpla la definición de subsecuencia creciente
  - Comparar cantidad de elementos en soluciones factibles

Otro problema de asignación

A diferencia del anterior, este es un problema de optimización

- También se puede ver como un CSP
- ... con el ingrediente adicional de ser un problema que busca minimizar

¿Existe subestructura óptima de forma intuitiva?

Posibles abordajes

- Backtracking:
  - Intentar tomar todas las subsecuencias
  - Verificar que con cada una cumpla la definición de subsecuencia creciente
  - Comparar cantidad de elementos en soluciones factibles
- Greedy

Una posible estrategia codiciosa

Una posible estrategia codiciosa

Considere un algoritmo "codicioso" que revisa de izquierda a derecha los elementos de la secuencia $A = \langle v_1, v_2, \ldots, v_n \rangle$ y determina si se debe incluir v_i con la siguiente estrategia codiciosa:

Una posible estrategia codiciosa

Considere un algoritmo "codicioso" que revisa de izquierda a derecha los elementos de la secuencia $A = \langle v_1, v_2, \ldots, v_n \rangle$ y determina si se debe incluir v_i con la siguiente estrategia codiciosa:

"*Si es menor que el último elemento incluido, no se le incluye.*
En otro caso, se incluye."

Una posible estrategia codiciosa

Considere un algoritmo "codicioso" que revisa de izquierda a derecha los elementos de la secuencia $A = \langle v_1, v_2, \ldots, v_n \rangle$ y determina si se debe incluir v_i con la siguiente estrategia codiciosa:

"*Si es menor que el último elemento incluido, no se le incluye. En otro caso, se incluye*."

¿Cómo demostramos que esta estrategia no entrega el óptimo?

Una posible estrategia codiciosa

Idea general para desmentir estrategias "*codiciosas falsas*"

Una posible estrategia codiciosa

Idea general para desmentir estrategias "*codiciosas falsas*"

- Debemos aprovechar la decisión de la estrategia

Una posible estrategia codiciosa

Idea general para desmentir estrategias "*codiciosas falsas*"

- Debemos aprovechar la decisión de la estrategia
- Forzar a que cometa un error basado en su decisión local

Una posible estrategia codiciosa

Idea general para desmentir estrategias "*codiciosas falsas*"

- Debemos aprovechar la decisión de la estrategia
- Forzar a que cometa un error basado en su decisión local

¿Qué secuencia muestra que esta estrategia no funciona?

Una posible estrategia codiciosa

Idea general para desmentir estrategias "*codiciosas falsas*"

- Debemos aprovechar la decisión de la estrategia
- Forzar a que cometa un error basado en su decisión local

¿Qué secuencia muestra que esta estrategia no funciona?

Podemos tomar cualquier secuencia cuyo primer elemento no debiera ser incluído: $A = \langle 10, 1, 2, 3, 4 \rangle$.

Sumario

Objetivos de la clase

Objetivos de la clase

- ☐ Comparar las técnicas estudiadas

Objetivos de la clase

- ☐ Comparar las técnicas estudiadas
- ☐ Aplicar las técnicas para resolver problemas