Programació i Algorísmia Avançada

Grau en Intel·ligència Artificial, FIB–UPC

José Luis Balcázar, Jordi Delgado

Dept. CS, UPC

2025–26, Quadrimestre de primavera

Contenido

Presentación

Búsqueda combinatoria

Teoría de lenguajes formales

Calculabilidad e indecidibilidad

Clases de complejidad

Contenido

Presentación

Búsqueda combinatoria

Teoría de lenguajes formales

Calculabilidad e indecidibilidad

Clases de complejidad

Algorítmica

En sentido laxo, incluye modelos abstractos de cálculo

- Búsqueda combinatoria: concepto; "backtracking"; esquemas "greedy"; Programación Dinámica; "divide-and-conquer".
- Lenguajes formales: gramáticas, modelos abstractos de cálculo.
- Computabilidad e indecidibilidad: las funciones recursivas parciales; lambda-cálculo.
- Teoría de la Complejidad; NP-completitud.

Lab

Y parte de los exámenes

He enviado a las direcciones de e-mail `estudiantat.upc.edu` invitaciones al curso de `jutge.org` con el mismo nombre que la asignatura. Quien no lo haya recibido que me avise por e-mail (a `jose.luis.balcazar@upc.edu`).

Hoy

Repasamos recursividad y árboles

- `https://jutge.org/problems/X91812`
- `https://jutge.org/problems/P63448`
- `https://jutge.org/problems/P90133`

Contenido

Presentación

Búsqueda combinatoria
- Búsqueda exhaustiva
- Estructura de subproblemas
- Backtracking
- Esquemas “greedy”
- Programación Dinámica (Dynamic Programming)

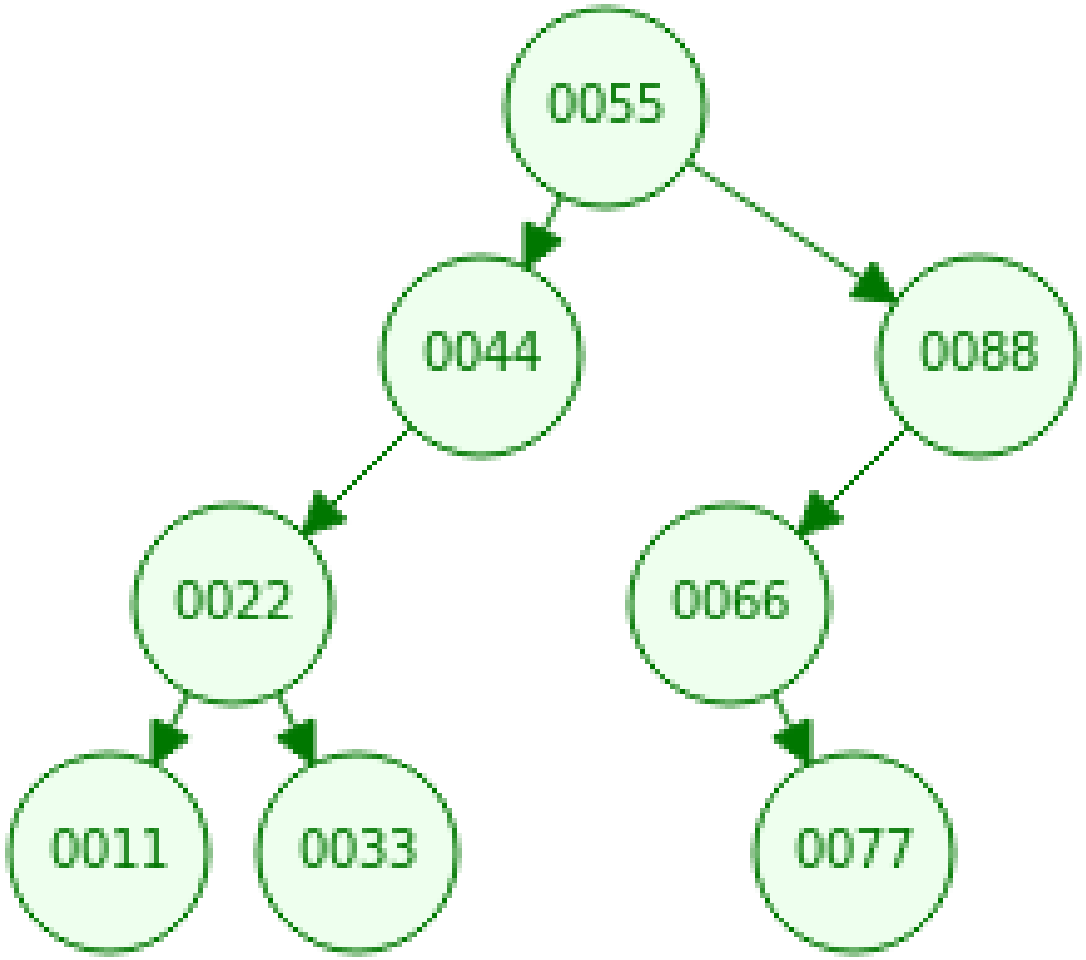
Teoría de lenguajes formales

Calculabilidad e indecidibilidad

Clases de complejidad

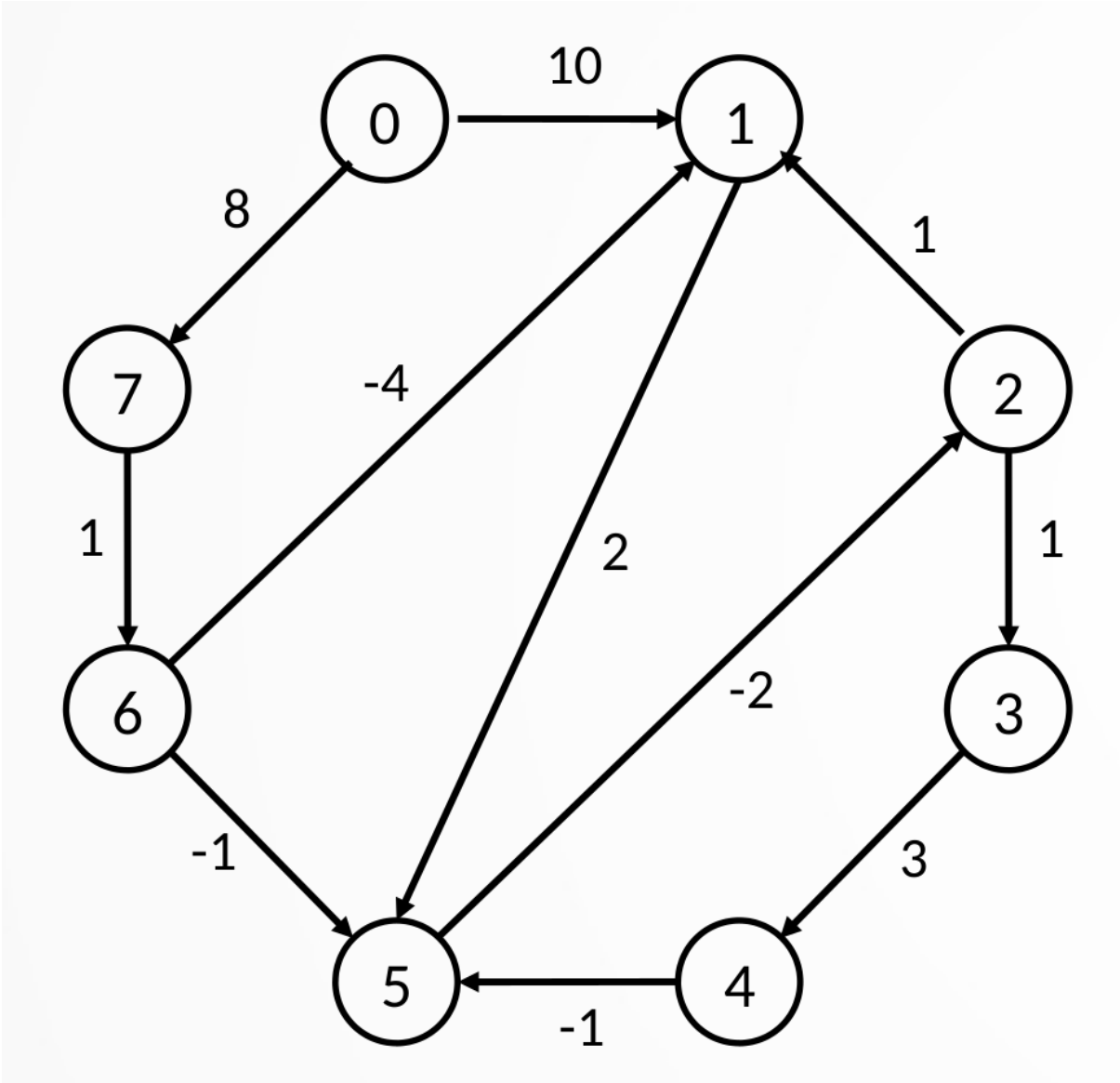
Recorridos de árboles

Repaso: preorden, inorden, postorden



Recorridos de grafos

Repaso: depth-first search



Búsqueda combinatoria, I

Combinatorial search

Algunas estrategias de diseño de algoritmos, de entre las muchísimas posibles, han resultado particularmente exitosas.

Contexto intuitivo para explicarlas

y analizar sus parecidos y diferencias:

- noción de "caso" de un problema computacional;
- noción de "solución candidata" para un caso;
- noción de "solución que buscamos", en dos posibles enfoques:

  (a) mera existencia (una solución? o todas ellas?),
  (b) optimalidad (maximización? minimización?).

Por supuesto, no todo problema computacional admite este tipo de análisis, pero muchos sí lo permiten (y aún más si tomamos estas nociones guía de forma un poco relajada pero aún útil).

Búsqueda combinatoria, II

Frecuentemente podemos aplicar el esquema de más de una manera

Árboles de expansión:

Dado un grafo conexo, con pesos en las aristas, encuéntrese en él un subgrafo conexo

(a) que conecta todos los vértices sin crear ciclos;

Búsqueda combinatoria, II

Frecuentemente podemos aplicar el esquema de más de una manera

Árboles de expansión:

Dado un grafo conexo, con pesos en las aristas, encuéntrese en él un subgrafo conexo

(a) que conecta todos los vértices sin crear ciclos; o

(b) que conecta todos los vértices con el mínimo peso total.

Búsqueda combinatoria, II

Frecuentemente podemos aplicar el esquema de más de una manera

Árboles de expansión:

Dado un grafo conexo, con pesos en las aristas, encuéntrese en él un subgrafo conexo

(a) que conecta todos los vértices sin crear ciclos; o

(b) que conecta todos los vértices con el mínimo peso total.

"Mochilas":

Dados números V y W y un conjunto de objetos, cada uno con un peso y un valor, encuéntrese un subconjunto de tales objetos

(a) que alcanza valor total al menos V pero pesa a lo más W;

(b) que alcanza el mayor valor posible pero pesa a lo más W;

(c) que alcanza valor total al menos V pero pesa lo menos posible.

Búsqueda combinatoria, III

O bien: Árboles de expansión, I

"Spanning trees":

- Noción de "caso" de un problema computacional:

  "Dado un grafo conexo con pesos en las aristas..."
- noción de "solución candidata" para un caso:

  "encuéntrese en él un subgrafo conexo que..."";
- noción de "solución que buscamos", en dos posibles enfoques:

  (a) mera existencia, como:
  "conecta todos los vértices sin crear ciclos" o

Búsqueda combinatoria, III

O bien: Árboles de expansión, I

"Spanning trees":

- Noción de "caso" de un problema computacional:

  "Dado un grafo conexo con pesos en las aristas..."
- noción de "solución candidata" para un caso:

  "encuéntrese en él un subgrafo conexo que..."
- noción de "solución que buscamos", en dos posibles enfoques:
  (a) mera existencia, como:
  "conecta todos los vértices sin crear ciclos" o
  (b) optimalidad (maximización o minimización), como:
  "conecta todos los vértices con el mínimo peso total".

Búsqueda combinatoria, IV

O bien: Mochila, I

"Mochilas":

- Noción de "caso" de un problema computacional:

  "Dados números V y W y un conjunto de objetos, cada uno con un peso y un valor..."

- noción de "solución candidata" para un caso:

  "encuéntrese un subconjunto de tales objetos..."

- noción de "solución que buscamos", en dos posibles enfoques:

  (a) mera existencia, como:
  "que alcanza valor total al menos V pero pesa a lo más W";

  (b) optimalidad (maximización o minimización), como:
  "que alcanza el mayor valor posible pero pesa a lo más W;"
  o:
  "que alcanza valor total al menos V pero pesa lo menos posible".

Empezaremos solucionando versiones decisionales, y luego extenderemos las soluciones a los casos de optimización.

Búsqueda exhaustiva, I

Simplemente "probemos todas las posibles soluciones", ¿no?

Al encontrarnos con un nuevo problema:

¿Cómo proceder?

1. Exploramos una o varias maneras de encajarlo en el esquema de búsqueda combinatoria.
2. De los esquemas algorítmicos que conozcamos, ¿cuáles podemos aplicar?
3. O... ¿tenemos que explorar todas las posibilidades?

Mochila, II

Versión decisional, buscaremos primero todas las soluciones

Dados:

- objetos $i \in \{0, \ldots, N-1\}$
- con pesos $w[i]$ y valores $v[i]$,
- máxima capacidad de la mochila W,
- valor total deseado V,

encuéntrese un conjunto de objetos "que poner en la mochila" de manera que:

- su peso total no supera la capacidad máxima W, y
- su valor total es al menos el valor deseado V.

Ejemplo:

Peso máximo $W = 26$, valor deseado $V = 45$ con objetos de:

Pesos:	9	8	12	11	7
Valores:	16	15	24	23	13

Mochila, III

"La cuenta de la vieja", "brute force", "perebor"

```
def slow_knapsack(objects, max_w, min_v)
    sols = list()
    for candidate in powerset(objects):
        if (totalweight(candidate) <= max_w and
            totalvalue(candidate) >= min_v):
            sols.append(candidate)
    return sols
```

Mochila, III

"La cuenta de la vieja", "brute force", "perebor"

```
def slow_knapsack(objects, max_w, min_v)
    sols = list()
    for candidate in powerset(objects):
        if (totalweight(candidate) <= max_w and
            totalvalue(candidate) >= min_v):
            sols.append(candidate)
    return sols
```

Diversas posibilidades para el iterador `powerset`:

- Resuélvelo sin particular inspiración previa: P18957.
- "Itertools recipes" en la documentación oficial de Python, capítulo sobre `itertools`.
- Aprender a programar generadores (es fácil) y hacerlos recursivos (ya no tan fácil).

Mochila, III

"La cuenta de la vieja", "brute force", "perebor"

```
def slow_knapsack(objects, max_w, min_v)
    sols = list()
    for candidate in powerset(objects):
        if (totalweight(candidate) <= max_w and
            totalvalue(candidate) >= min_v):
            sols.append(candidate)
    return sols
```

Diversas posibilidades para el iterador `powerset`:

- Resuélvelo sin particular inspiración previa: P18957.
- "Itertools recipes" en la documentación oficial de Python, capítulo sobre `itertools`.
- Aprender a programar generadores (es fácil) y hacerlos recursivos (ya no tan fácil).
- Demasiado lento para casi cualquier propósito práctico.

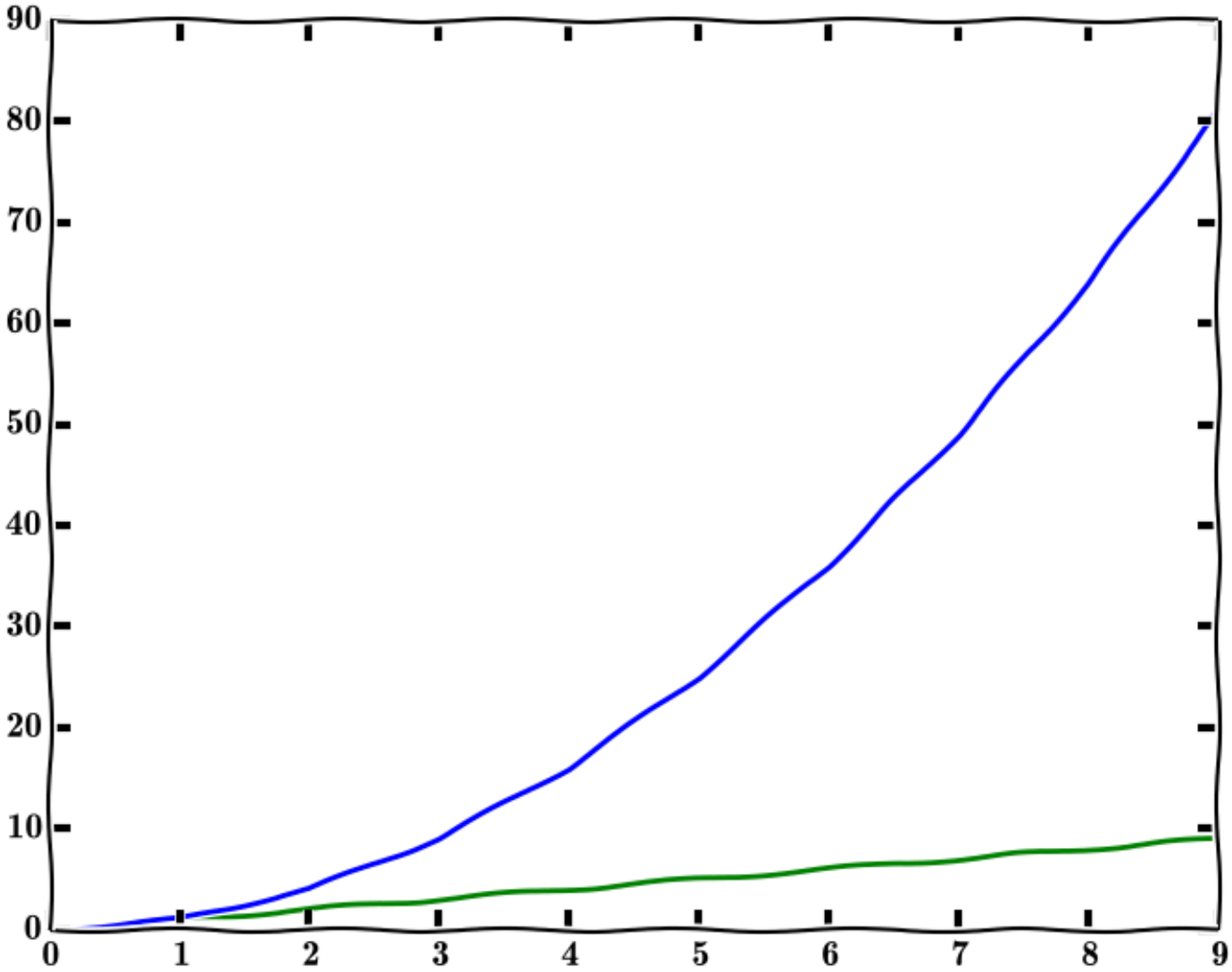
Búsqueda exhaustiva, II

Límites de la búsqueda exhaustiva

1. "Set-based combinatorial search": buscamos una aguja en el pajar de todos los subconjuntos de un conjunto dado.
2. "Permutation-based combinatorial search": buscamos una aguja en el pajar de todas las permutaciones de una secuencia dada.
3. ¿Realmente tenemos que explorar todas las posibilidades?
   - Todos los subconjuntos ("powerset")... 2^N casos.
   - Todas las permutaciones... $N!$ casos.

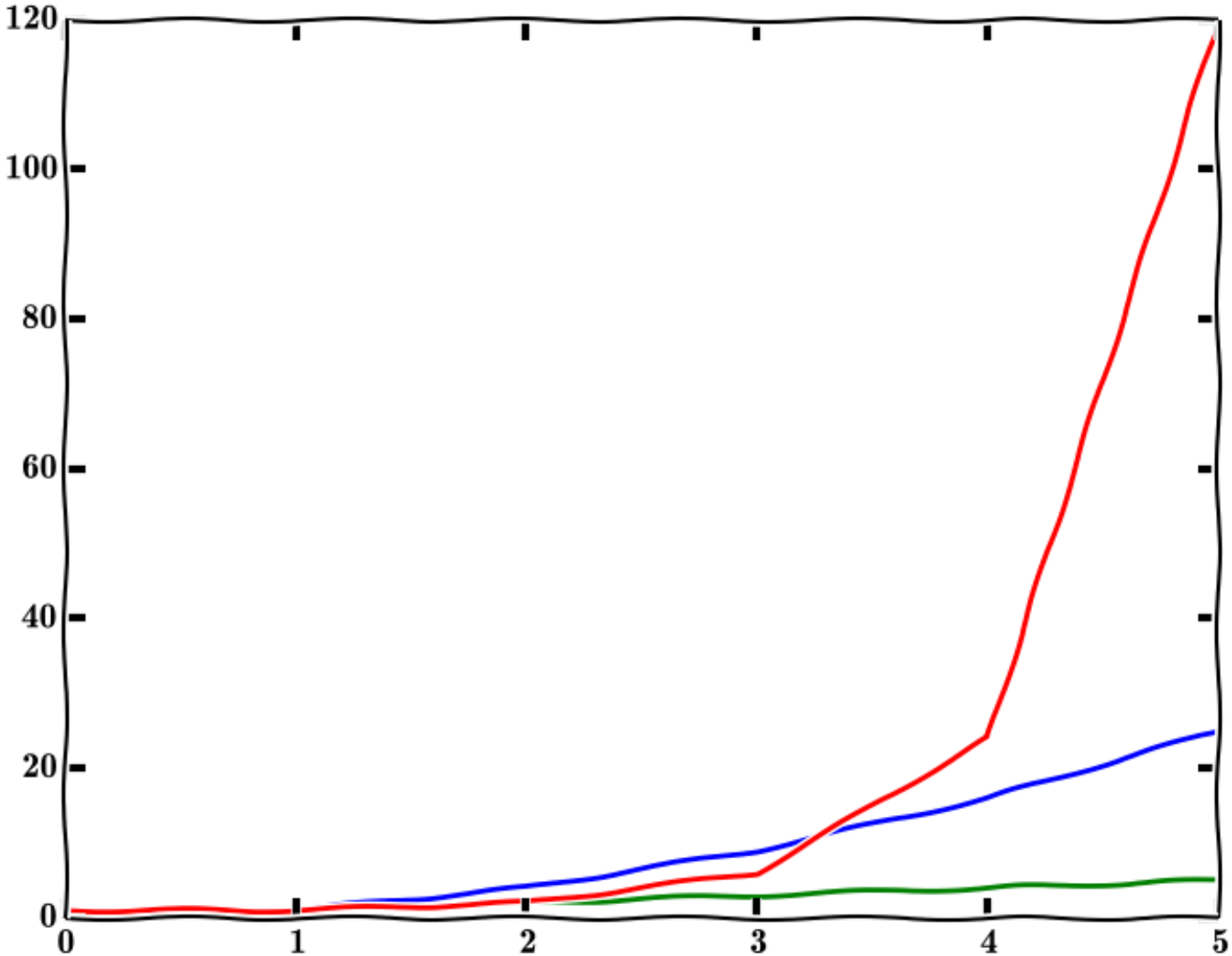
El factorial y el crecimiento exponential, I

No tomarás el nombre de la Exponencial en vano!



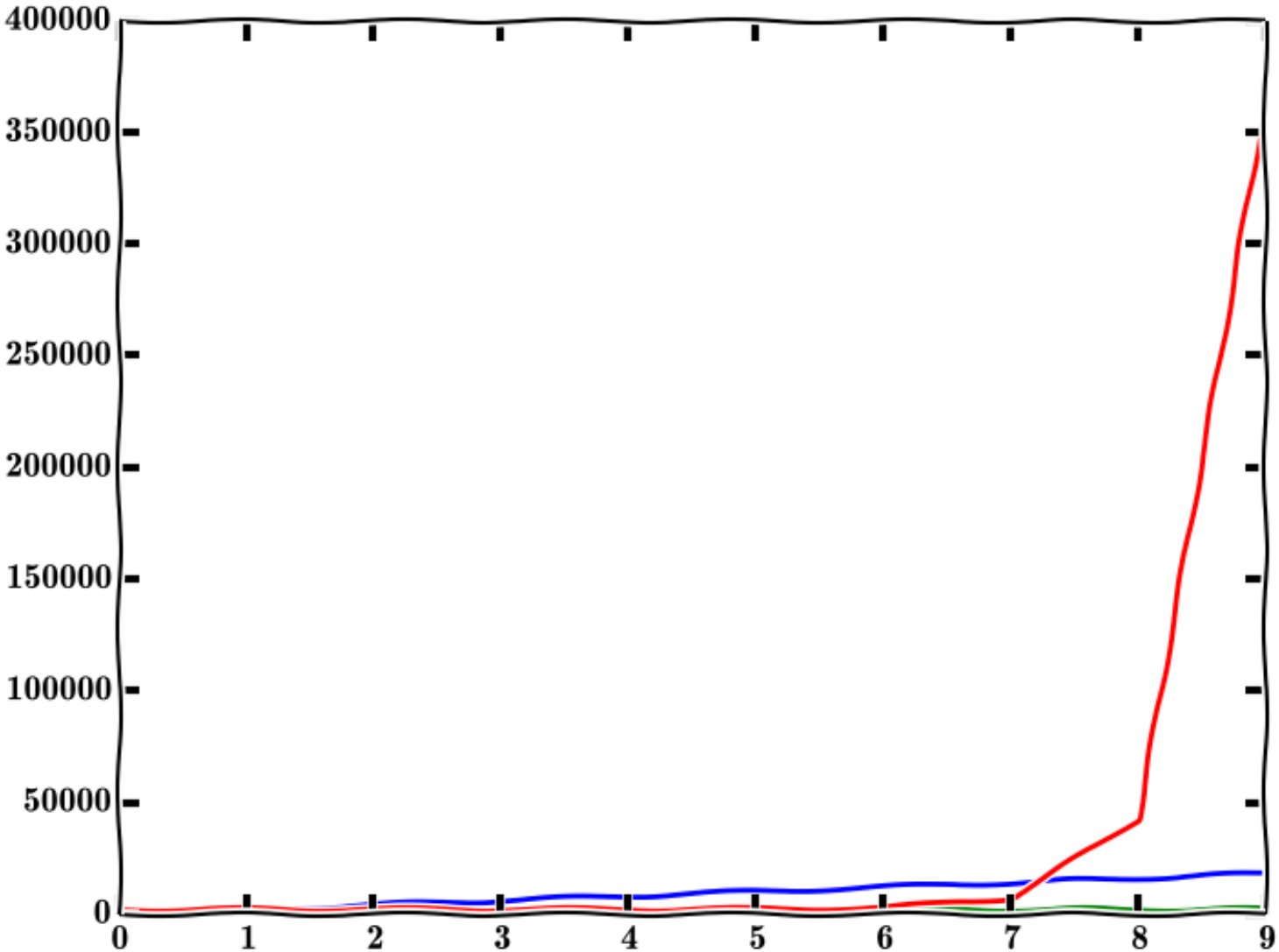
El factorial y el crecimiento exponential, I

No tomarás el nombre de la Exponencial en vano!



El factorial y el crecimiento exponential, I

No tomarás el nombre de la Exponencial en vano!



El factorial y el crecimiento exponential, II

$N!$ crece exponencialmente, dijo Stirling

Supongamos:

- que sólo necesitamos una operación elemental por cada una de las $N!$ configuraciones posibles, y que
- podemos contar con que se realicen 13000 billones de operaciones por segundo (13×10^{15}).

Entonces, tardaremos:

para $N = 12$: milmillonésimas de segundo (10^{-9});

El factorial y el crecimiento exponential, II

$N!$ crece exponencialmente, dijo Stirling

Supongamos:

- que sólo necesitamos una operación elemental por cada una de las $N!$ configuraciones posibles, y que
- podemos contar con que se realicen 13000 billones de operaciones por segundo (13×10^{15}).

Entonces, tardaremos:

para $N = 12$: milmillonésimas de segundo (10^{-9});

para $N = 15$: 8 milésimas de segundo (8×10^{-3});

El factorial y el crecimiento exponential, II

$N!$ crece exponencialmente, dijo Stirling

Supongamos:

- que sólo necesitamos una operación elemental por cada una de las $N!$ configuraciones posibles, y que
- podemos contar con que se realicen 13000 billones de operaciones por segundo (13×10^{15}).

Entonces, tardaremos:

para $N = 12$: milmillonésimas de segundo (10^{-9});

para $N = 15$: 8 milésimas de segundo (8×10^{-3});

para $N = 18$: medio segundo;

El factorial y el crecimiento exponential, II

$N!$ crece exponencialmente, dijo Stirling

Supongamos:

- que sólo necesitamos una operación elemental por cada una de las $N!$ configuraciones posibles, y que
- podemos contar con que se realicen 13000 billones de operaciones por segundo (13×10^{15}).

Entonces, tardaremos:

para $N = 12$: milmillonésimas de segundo (10^{-9});

para $N = 15$: 8 milésimas de segundo (8×10^{-3});

para $N = 18$: medio segundo;

para $N = 21$: una hora;

El factorial y el crecimiento exponential, II

$N!$ crece exponencialmente, dijo Stirling

Supongamos:

- que sólo necesitamos una operación elemental por cada una de las $N!$ configuraciones posibles, y que
- podemos contar con que se realicen 13000 billones de operaciones por segundo (13×10^{15}).

Entonces, tardaremos:

para $N = 12$: milmillonésimas de segundo (10^{-9});

para $N = 15$: 8 milésimas de segundo (8×10^{-3});

para $N = 18$: medio segundo;

para $N = 21$: una hora;

para $N = 24$: un año y medio;

El factorial y el crecimiento exponential, II

$N!$ crece exponencialmente, dijo Stirling

Supongamos:

- que sólo necesitamos una operación elemental por cada una de las $N!$ configuraciones posibles, y que
- podemos contar con que se realicen 13000 billones de operaciones por segundo (13×10^{15}).

Entonces, tardaremos:

para $N = 12$: milmillonésimas de segundo (10^{-9});

para $N = 15$: 8 milésimas de segundo (8×10^{-3});

para $N = 18$: medio segundo;

para $N = 21$: una hora;

para $N = 24$: un año y medio;

para $N = 27$: más de 250 siglos...

Búsqueda combinatoria, V

Variantes

¿Qué estructura combinatoria hay detrás?

- ¿Conjuntos? "Set-based backtracking".
- ¿Permutaciones? "Permutation-based backtracking".
- ...

Y, en términos de los detalles algorítmicos, puede ser:

- Recorrido para encontrar todas las soluciones.

Búsqueda combinatoria, V

Variantes

¿Qué estructura combinatoria hay detrás?

- ¿Conjuntos? "Set-based backtracking".
- ¿Permutaciones? "Permutation-based backtracking".
- ...

Y, en términos de los detalles algorítmicos, puede ser:

- Recorrido para encontrar todas las soluciones.
- Búsqueda (similar a la lineal) para encontrar una solución.

Búsqueda combinatoria, V

Variantes

¿Qué estructura combinatoria hay detrás?

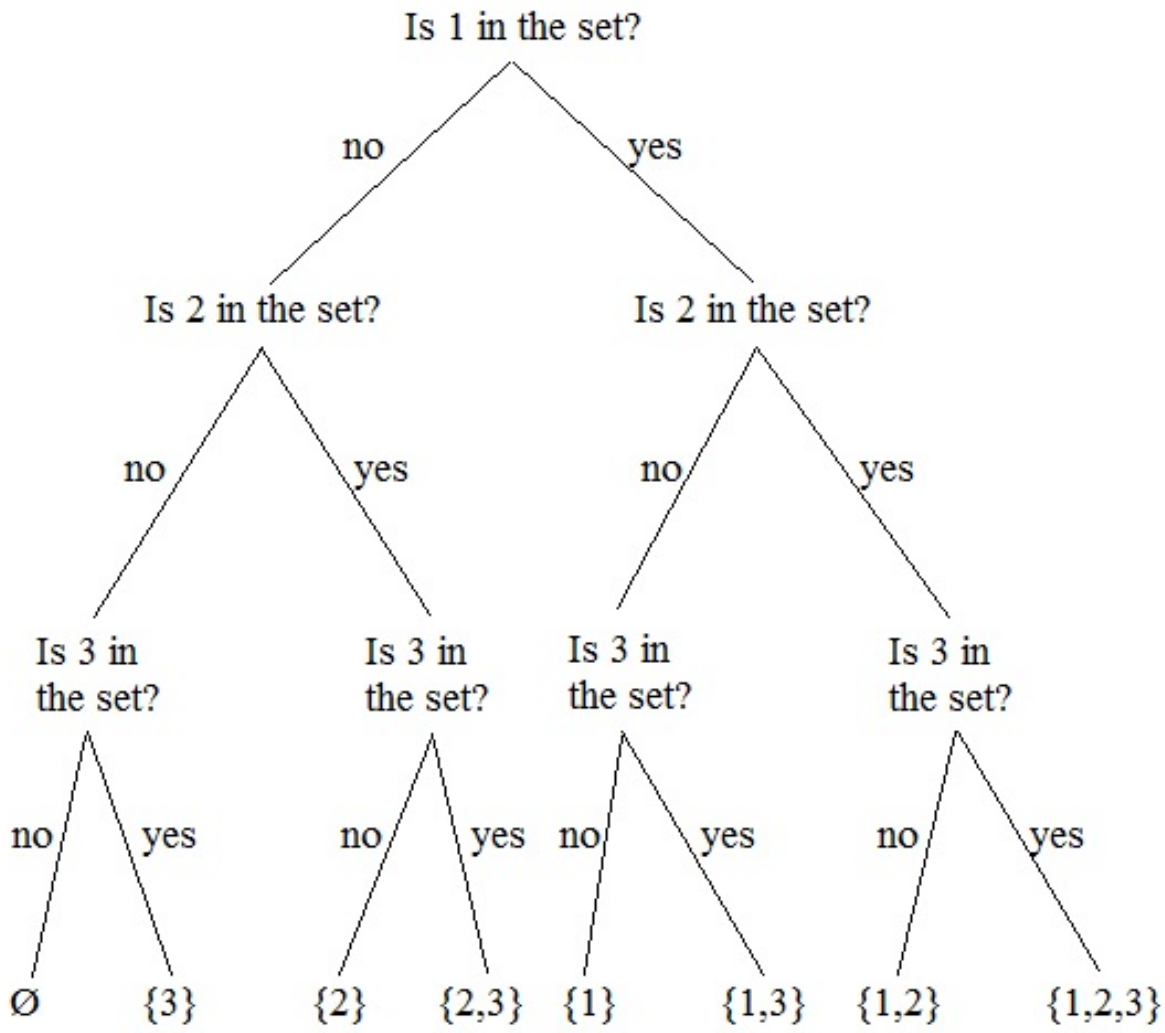
- ¿Conjuntos? "Set-based backtracking".
- ¿Permutaciones? "Permutation-based backtracking".
- ...

Y, en términos de los detalles algorítmicos, puede ser:

- Recorrido para encontrar todas las soluciones.
- Búsqueda (similar a la lineal) para encontrar una solución.
- Recorrido de optimización para encontrar la mejor solución.

Mochila, IV

Recorrido alternativo del "powerset"



By: Brian M. Scott at math.stackexchange.com

Mochila, V

"La cuenta de la vieja" siguiendo el recorrido alternativo

```
def knapsack(weights, values, current_item, max_w, min_v):
  if current_item == -1:
    "all items considered, none left"
    if min_v <= 0 and max_w >= 0:
      return [ list() ]
    else:
      return list()

  sols0 = knapsack(weights, values, current_item - 1,
                   max_w, min_v)
  sols1 = knapsack(weights, values, current_item - 1,
                   max_w - weights[current_item],
                   min_v - values[current_item])
  sols0.extend( sol + [ current_item ] for sol in sols1 )
  return sols0
```

Búsqueda combinatoria, VI

El siguiente ingrediente

Adicionalmente:

Los candidatos a solución se estructuran en

- una noción de "subproblema", obtenido a través de una "secuencia de decisiones" que progresan hacia las soluciones candidatas

Búsqueda combinatoria, VI

El siguiente ingrediente

Adicionalmente:

Los candidatos a solución se estructuran en

- una noción de "subproblema", obtenido a través de una "secuencia de decisiones" que progresan hacia las soluciones candidatas

  (frecuentemente se denomina "problemas locales" a los subproblemas, y entonces el problema original se denomina "global");

Búsqueda combinatoria, VI

El siguiente ingrediente

Adicionalmente:

Los candidatos a solución se estructuran en

- una noción de "subproblema", obtenido a través de una "secuencia de decisiones" que progresan hacia las soluciones candidatas

  (frecuentemente se denomina "problemas locales" a los subproblemas, y entonces el problema original se denomina "global");
- una función que nos indica si una secuencia de decisiones es

Búsqueda combinatoria, VI

El siguiente ingrediente

Adicionalmente:

Los candidatos a solución se estructuran en

- una noción de "subproblema", obtenido a través de una "secuencia de decisiones" que progresan hacia las soluciones candidatas

  (frecuentemente se denomina "problemas locales" a los subproblemas, y entonces el problema original se denomina "global");
- una función que nos indica si una secuencia de decisiones es
  (a) ya "inaceptable"

Búsqueda combinatoria, VI

El siguiente ingrediente

Adicionalmente:

Los candidatos a solución se estructuran en

- una noción de "subproblema", obtenido a través de una "secuencia de decisiones" que progresan hacia las soluciones candidatas

  (frecuentemente se denomina "problemas locales" a los subproblemas, y entonces el problema original se denomina "global");
- una función que nos indica si una secuencia de decisiones es
  (a) ya "inaceptable"
  (es decir, el subproblema no tiene solución) o

Búsqueda combinatoria, VI

El siguiente ingrediente

Adicionalmente:

Los candidatos a solución se estructuran en

- una noción de "subproblema", obtenido a través de una "secuencia de decisiones" que progresan hacia las soluciones candidatas

  (frecuentemente se denomina "problemas locales" a los subproblemas, y entonces el problema original se denomina "global");
- una función que nos indica si una secuencia de decisiones es
  (a) ya "inaceptable"
  (es decir, el subproblema no tiene solución) o
  (b) "aceptable" pero aún "incompleta"

Búsqueda combinatoria, VI

El siguiente ingrediente

Adicionalmente:

Los candidatos a solución se estructuran en

- una noción de "subproblema", obtenido a través de una "secuencia de decisiones" que progresan hacia las soluciones candidatas

  (frecuentemente se denomina "problemas locales" a los subproblemas, y entonces el problema original se denomina "global");
- una función que nos indica si una secuencia de decisiones es
  (a) ya "inaceptable"
  (es decir, el subproblema no tiene solución) o
  (b) "aceptable" pero aún "incompleta"
  (puede ser que el problema tenga solución, hay que continuar la exploración) o

Búsqueda combinatoria, VI

El siguiente ingrediente

Adicionalmente:

Los candidatos a solución se estructuran en

- una noción de "subproblema", obtenido a través de una "secuencia de decisiones" que progresan hacia las soluciones candidatas

  (frecuentemente se denomina "problemas locales" a los subproblemas, y entonces el problema original se denomina "global");
- una función que nos indica si una secuencia de decisiones es
  (a) ya "inaceptable"
  (es decir, el subproblema no tiene solución) o
  (b) "aceptable" pero aún "incompleta"
  (puede ser que el problema tenga solución, hay que continuar la exploración) o
  (c) una solución "completa" para el problema global.

Backtracking, I

Concepto

Organizamos la exploración de manera controlada:

Depth-First Search / preorden, excepto que el grafo o árbol es implícito.

- Cada subproblema es un vértice de un grafo o árbol (probablemente muy grande) que queda en nuestra imaginación.
- Las aristas de ese grafo imaginario son decisiones que nos llevan de un subproblema a otro.
- Y lo principal: cuando detectamos un subproblema no factible ("callejón sin salida"), nos ahorramos la exploración de todas las configuraciones que requieran solucionarlo.

(El nombre viene "heredado" de antaño, antes de que la programación recursiva fuera una opción generalizada: era preciso "programar explícitamente" el cambio de subárbol a explorar.)

Búsqueda combinatoria, VII

O bien: Árboles de expansión, II

Árboles de expansión:

Subproblema:

- encontrar el árbol de expansión de un subgrafo, o bien
- completar un único árbol de expansión incompleto:
  - manteniendo un árbol parcial ya construido, o bien

Búsqueda combinatoria, VII

O bien: Árboles de expansión, II

Árboles de expansión:

Subproblema:

- encontrar el árbol de expansión de un subgrafo, o bien
- completar un único árbol de expansión incompleto:
  - manteniendo un árbol parcial ya construido, o bien
  - manteniendo un conjunto de árboles parciales ya construidos (spanning forest)...

Búsqueda combinatoria, VII

O bien: Árboles de expansión, II

Árboles de expansión:

Subproblema:

- encontrar el árbol de expansión de un subgrafo, o bien
- completar un único árbol de expansión incompleto:
  - manteniendo un árbol parcial ya construido, o bien
  - manteniendo un conjunto de árboles parciales ya construidos (spanning forest)...

Secuencia de decisiones: el árbol crece en una arista más...

(a) solución "completa" para el problema global: conecta todo,

Búsqueda combinatoria, VII

O bien: Árboles de expansión, II

Árboles de expansión:

Subproblema:

- encontrar el árbol de expansión de un subgrafo, o bien
- completar un único árbol de expansión incompleto:
  - manteniendo un árbol parcial ya construido, o bien
  - manteniendo un conjunto de árboles parciales ya construidos (spanning forest)...

Secuencia de decisiones: el árbol crece en una arista más...

(a) solución "completa" para el problema global: conecta todo,

(b) ya "inaceptable": la nueva arista crea un ciclo,

Búsqueda combinatoria, VII

O bien: Árboles de expansión, II

Árboles de expansión:

Subproblema:

- encontrar el árbol de expansión de un subgrafo, o bien
- completar un único árbol de expansión incompleto:
  - manteniendo un árbol parcial ya construido, o bien
  - manteniendo un conjunto de árboles parciales ya construidos (spanning forest)...

Secuencia de decisiones: el árbol crece en una arista más...

(a) solución "completa" para el problema global: conecta todo,

(b) ya "inaceptable": la nueva arista crea un ciclo,

(c) "aceptable" pero aún "incompleta": todos los demás casos.

Búsqueda combinatoria, VIII

O bien: Mochila, VI

¿Podemos lograr valor total al menos V con peso no superior a W?

Subproblema: consideramos sólo un subconjunto de los objetos.

Búsqueda combinatoria, VIII

O bien: Mochila, VI

¿Podemos lograr valor total al menos V con peso no superior a W?

Subproblema: consideramos sólo un subconjunto de los objetos.

Secuencia de decisiones: consideramos un objeto nuevo; o bien nos lo quedamos o bien lo descartamos.

(a) solución "completa" para el problema global: hemos considerado todos los objetos;

Búsqueda combinatoria, VIII

O bien: Mochila, VI

¿Podemos lograr valor total al menos V con peso no superior a W?

Subproblema: consideramos sólo un subconjunto de los objetos.

Secuencia de decisiones: consideramos un objeto nuevo; o bien nos lo quedamos o bien lo descartamos.

(a) solución "completa" para el problema global: hemos considerado todos los objetos;

(b) ya "inaceptable": el nuevo peso total supera W y no decrecerá al añadir más objetos;

Búsqueda combinatoria, VIII

O bien: Mochila, VI

¿Podemos lograr valor total al menos V con peso no superior a W?

Subproblema: consideramos sólo un subconjunto de los objetos.

Secuencia de decisiones: consideramos un objeto nuevo; o bien nos lo quedamos o bien lo descartamos.

(a) solución "completa" para el problema global: hemos considerado todos los objetos;

(b) ya "inaceptable": el nuevo peso total supera W y no decrecerá al añadir más objetos;

(c) "aceptable" pero aún "incompleta": todos los demás casos.

Mochila, VII

Aplicando backtracking

```
def knapsack(weights, values, current_item, max_w, min_v):
  if current_item == -1:
    "all items considered, none left"
    if min_v <= 0:
      return [ list() ]
    else:
      return list()
  sols0 = knapsack(weights, values, current_item - 1,
                   max_w, min_v)
  if weights[current_item] <= max_w:
    "current_item >= 0 is a valid item to consider next"
    sols1 = knapsack(weights, values, current_item - 1,
                     max_w - weights[current_item],
                     min_v - values[current_item])
    sols0.extend(sol + [ current_item ] for sol in sols1)
  return sols0
```

Mochila, VIII

Sofisticaciones

En este ejemplo, la secuencia de decisiones que lleva al punto en que estamos se reduce a los nuevos valores de `max_w` y `min_v`. No es buena inspiración para problemas en que sea preciso tener en cuenta las decisiones ya tomadas.

Mochila, VIII

Sofisticaciones

En este ejemplo, la secuencia de decisiones que lleva al punto en que estamos se reduce a los nuevos valores de `max_w` y `min_v`. No es buena inspiración para problemas en que sea preciso tener en cuenta las decisiones ya tomadas.

¿Cómo podemos solucionarlo manteniendo explícitamente las decisiones tomadas?

Mochila, IX

Búsqueda exhaustiva con candidato explícito

```
def knapsack(weights, values, current_item,
             max_w, min_v, cand, cand_w, cand_v):
  if current_item == -1:
    if cand_v >= min_v and cand_w <= max_w:
      return [ cand ]
    else:
      return list()
  else:
    sols = knapsack(weights, values, current_item - 1,
                    max_w, min_v, cand, cand_w, cand_v)
    sols.extend(knapsack(weights, values, current_item-1,
                    max_w, min_v,
                    cand + [ current_item ],
                    cand_w + weights[current_item],
                    cand_v + values[current_item]))
    return sols
```

Mochila, X

Backtracking con candidato explícito

```
def knapsack(weights, values, current_item, max_w, min_v,
             cand, cand_w, cand_v):
  if current_item == -1:
    if cand_v >= min_v and cand_w <= max_w:
      return [ cand ]
    else:
      return list()
  else:
    sols = knapsack(weights, values, current_item - 1,
                    max_w, min_v, cand, cand_w, cand_v)
    if weights[current_item] <= max_w:
     sols.extend(knapsack(weights, values, current_item-1,
                    max_w, min_v,
                    cand + [ current_item ],
                    cand_w + weights[current_item],
                    cand_v + values[current_item]))
    return sols
```

Mochila, XI

Backtracking con candidato explícito, evitando copias

```
def knapsack(weights, values, current_item, max_w, min_v,
             cand, cand_w, cand_v):
  if current_item == -1:
    if cand_v >= min_v and cand_w <= max_w:
      return [ cand.copy() ]
    else: return list()
  else:
    sols = knapsack(weights, values, current_item - 1,
                    max_w, min_v, cand, cand_w, cand_v)
    if weights[current_item] <= max_w:
     cand.append(current_item)
     sols.extend(knapsack(weights, values, current_item-1,
                    max_w, min_v, cand,
                    cand_w + weights[current_item],
                    cand_v + values[current_item]))
     cand.pop() # backtracking happens here!
    return sols
```

Búsqueda combinatoria, IX

Existencia versus optimización

En el caso de problemas de optimización

(sea maximización o minimización) precisamos además

una función objetivo a optimizar,

- definida sobre candidatos a solución, pero
- de tal manera que se pueda extender de forma natural a los subproblemas locales (secuencias de decisiones).

Búsqueda combinatoria, X

O bien: Árboles de expansión, III

Árboles de expansión:

Secuencia de decisiones: el árbol crece en una arista más...

(a) ya "inaceptable": la nueva arista crea un ciclo,

(b) solución "completa" para el problema global: conecta todo,

(c) "aceptable" pero aún "incompleta": todos los demás casos.

Función objetivo:

- peso del árbol parcial en curso?

Búsqueda combinatoria, X

O bien: Árboles de expansión, III

Árboles de expansión:

Secuencia de decisiones: el árbol crece en una arista más...

(a) ya "inaceptable": la nueva arista crea un ciclo,

(b) solución "completa" para el problema global: conecta todo,

(c) "aceptable" pero aún "incompleta": todos los demás casos.

Función objetivo:

- peso del árbol parcial en curso?
- mejor peso posible para un árbol de expansión completo que extienda el árbol parcial en curso?

Búsqueda combinatoria, XI

O bien: Mochila, XII

Mochila, versión de optimización:

Lograr el máximo valor total con peso no superior a W.

Subproblema: consideramos sólo un subconjunto de los objetos.

Búsqueda combinatoria, XI

O bien: Mochila, XII

Mochila, versión de optimización:

Lograr el máximo valor total con peso no superior a W.

Subproblema: consideramos sólo un subconjunto de los objetos.

Secuencia de decisiones: consideramos un objeto nuevo; o bien nos lo quedamos o bien lo descartamos.

(a) ya "inaceptable": el nuevo peso total supera W;

(b) solución "completa" para el problema global: hemos considerado todos los objetos;

(c) "aceptable" pero aún "incompleta": todos los demás casos.

Búsqueda combinatoria, XI

O bien: Mochila, XII

Mochila, versión de optimización:

Lograr el máximo valor total con peso no superior a W.

Subproblema: consideramos sólo un subconjunto de los objetos.

Secuencia de decisiones: consideramos un objeto nuevo; o bien nos lo quedamos o bien lo descartamos.

(a) ya "inaceptable": el nuevo peso total supera W;

(b) solución "completa" para el problema global: hemos considerado todos los objetos;

(c) "aceptable" pero aún "incompleta": todos los demás casos.

Función objetivo:

- valor de la mochila en curso?

Búsqueda combinatoria, XII

O bien: Mochila, XIII

Mochila, enfoque alternativo:

Lograr el menor peso posible con un valor de al menos V.

Búsqueda combinatoria, XII

O bien: Mochila, XIII

Mochila, enfoque alternativo:

Lograr el menor peso posible con un valor de al menos V.

Subproblema: dado el conjunto de objetos aún no descartados, descartar nuevos objetos.

(Lector: complete el esquema por su propia cuenta.)

Mochila, XIV

Problema de optimización por búsqueda exhaustiva

```
def slow_knapsack(weights, values, itq, limw):
  mx = 0
  best = None
  for cand in powerset(range(itq)):
    if total(weights, cand) <= limw:
      cmx = total(values, cand)
      if cmx > mx:
        best = cand
        mx = cmx
  return best, total(weights, best), total(values, best)
```

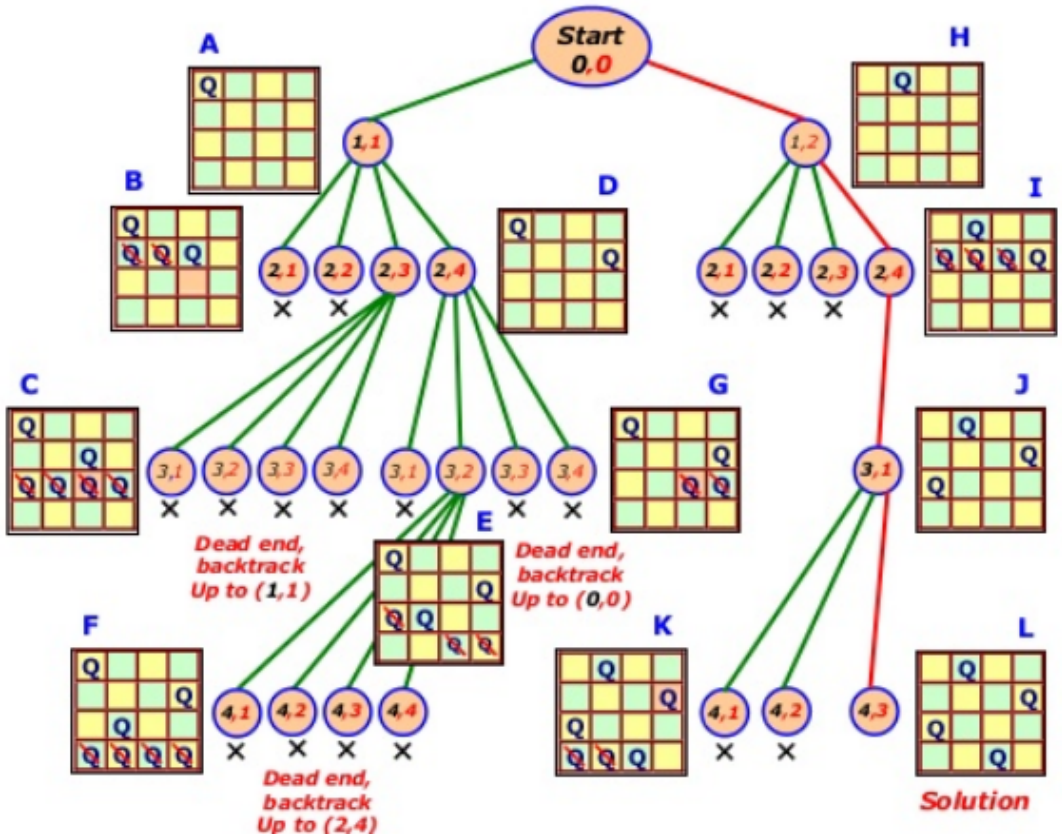
Mochila, XV

Problema de optimización por backtracking

```
def knapsack(weights, values, current_item, max_w):
  if current_item == -1:
    return ([],0,0)
  else:
    "current_item >= 0"
    best0, bestw0, bestv0 = knapsack(weights, values,
          current_item - 1, max_w)
    if weights[current_item] <= max_w:
      best1, bestw1, bestv1 = knapsack(weights, values,
          current_item - 1, max_w - weights[current_item])
      if bestv1 + values[current_item] > bestv0:
        best1.append(current_item)
        return (best1, bestw1 + weights[current_item],
                bestv1 + values[current_item])
  return best0, bestw0, bestv0
```

Ejemplo: *N-queens*, I

El árbol implícito: parte explorada hasta la primera solución



Fuente: `https://www.slideshare.net/praveenkumar33449138/02-problem-solvingsearchcontrol`

Ejemplo: *N-queens*, II

Busca todas las soluciones

```
def attempt(row, board, size):
  if row == size:
    board.draw()
  else:
    for column in range(size):
      if board.free(row, column):
        board.put_q(row, column)
        attempt(row + 1, board, size)
        board.remove_q(row, column)
```

Llamada inicial:

```
board = Board()
size = int(input("How many queens? "))
attempt(0, board, size)
```

Ejemplo: *N-queens*, III

Busca una solución

```
def attempt(row, board, size):
  if row == size:
    return True
  else:
    for column in range(size):
      if board.free(row, column):
        board.put_q(row, column)
        s = attempt(row + 1, board, size)
        if s:
          return True
        else:
          board.remove_q(row, column)
    return False
```

Ejemplo: *N-queens*, III

Busca una solución

```
def attempt(row, board, size):
  if row == size:
    return True
  else:
    for column in range(size):
      if board.free(row, column):
        board.put_q(row, column)
        s = attempt(row + 1, board, size)
        if s:
          return True
        else:
          board.remove_q(row, column)
    return False
```

Llamada inicial: declara el tablero, lee el tamaño, y llama así:

```
if attempt(0, board, size):
  board.draw()
```

Ejemplo: *N-queens*, IV

Por supuesto, podemos hacerlo mejor

Ideas a explorar:

- Simetrías: evita explorar una configuración que es, en esencia, "la misma" que una ya explorada.
- Adapta el orden en que se exploran las casillas de la fila en curso:
  - Cada casilla, si la usamos, ¿en cuánto nos reduce las posibilidades en las filas siguientes?
  - Exploramos primero las casillas que nos dejan más libertad para las filas siguientes, y dejamos las más restrictivas para después ("best-first search").
- ...

"Graph Colorability"

Dos variantes, sólo estudiamos una

"Vertex coloring":

Dado un grafo, asígnese un color a cada vértice de manera que no haya ninguna arista que conecte dos vértices del mismo color.

`http://mathworld.wolfram.com/images/eps-gif/VertexColoring_750.gif`

"Edge coloring":

Dado un grafo, asígnese un color a cada arista de manera que no haya ningún vértice en que confluyan dos o más aristas del mismo color.

`http://mathworld.wolfram.com/images/eps-gif/EdgeColoring_850.gif`

Hoy: "edge coloring".

Ejemplo: "3-colorability" en grafos 3-regulares, I

El árbol implícito: grafos con más y más aristas ya coloreadas

Restricción

Hoy, sólo grafos 3-regulares: todos los vértices tienen grado 3.

Enunciado:

Dado un grafo 3-regular G, asígnense colores a las aristas usando tres colores de manera que en cada vértice haya una arista de cada color.

Ideas para un esquema de "backtracking":

- Cada vértice del grafo implícito corresponde al grafo G con parte de las aristas ya coloreadas.

Ejemplo: "3-colorability" en grafos 3-regulares, I

El árbol implícito: grafos con más y más aristas ya coloreadas

Restricción

Hoy, sólo grafos 3-regulares: todos los vértices tienen grado 3.

Enunciado:

Dado un grafo 3-regular G, asígnense colores a las aristas usando tres colores de manera que en cada vértice haya una arista de cada color.

Ideas para un esquema de "backtracking":

- Cada vértice del grafo implícito corresponde al grafo G con parte de las aristas ya coloreadas.
- Vecinos de un vértice del grafo implícito: una arista más de G recibe color.

Ejemplo: "3-colorability" en grafos 3-regulares, I

El árbol implícito: grafos con más y más aristas ya coloreadas

Restricción

Hoy, sólo grafos 3-regulares: todos los vértices tienen grado 3.

Enunciado:

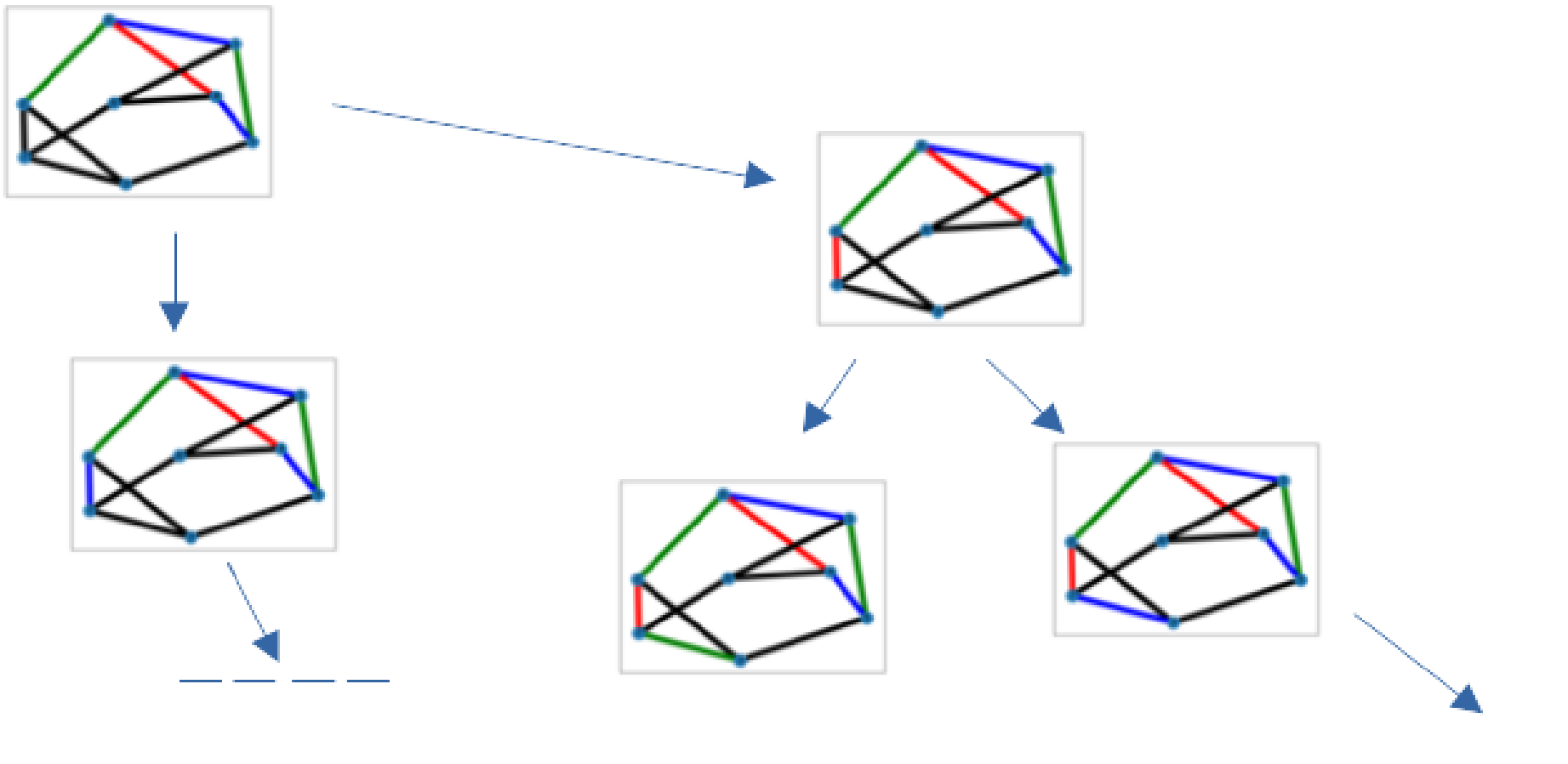
Dado un grafo 3-regular G, asígnense colores a las aristas usando tres colores de manera que en cada vértice haya una arista de cada color.

Ideas para un esquema de "backtracking":

- Cada vértice del grafo implícito corresponde al grafo G con parte de las aristas ya coloreadas.
- Vecinos de un vértice del grafo implícito: una arista más de G recibe color.
- ¿Cuál? Queremos asegurar que el grafo implícito es un árbol para evitar subproblemas repetidos.

Ejemplo: “3-colorability” en grafos 3-regulares, II

El árbol implícito (fragmento)



Ejemplo: "3-colorability" en grafos 3-regulares, III

Una opción de entre varias

Forzamos un orden sobre las aristas

y lo mantenemos estrictamente: si un camino del grafo implícito colorea primero la arista e_1 de G y después la arista e_2 de G, lo mismo ocurre en todos los caminos.

- Por ejemplo, "depth-first search" sobre G para marcar el orden.
- Eso asegura que, al colorear cada arista, al menos uno de los extremos ya ha gastado al menos un color.
- Además, el grafo implícito es un árbol: cada posible subproblema sólo se puede alcanzar de una manera.

Ejemplo: “3-colorability” en grafos 3-regulares, IV

Demo!

Basada en NetworkX y GraphViz:

- fijamos un orden de las aristas mediante la implementación de “depth-first search” de NetworkX;
- mantenemos el conjunto de colores disponibles en cada vértice;
- los vamos probando uno a uno y, con cada uno, lanzamos la llamada recursiva;
- Callejones sin salida: aristas para las que ya no quedan colores factibles.

Parafernalia adicional para informar de lo que va pasando y dibujar los grafos

(como el `dict gd` que mantiene el “layout” de GraphViz).

Ejemplo: "3-colorability" en grafos 3-regulares, V

El programa

```
def tricolor(g, edgelist):
  if not edgelist: return True
  else:
    u, v = edgelist.pop()
    possib = g.node[u]['free'] & g.node[v]['free']
    for c in possib:
      g.edges[u, v]['color'] = c
      g.nodes[u]['free'].remove(c)
      g.nodes[v]['free'].remove(c)
      success = tricolor(g, edgelist)
      if success: return True
      # else, free again the colors, try next possib
      g.edges[u, v]['color'] = noncolor
      g.nodes[u]['free'].add(c)
      g.nodes[v]['free'].add(c)
    edgelist.append((u, v)) # backtrack!
    return False
```

Ejemplo: "3-colorability" en grafos 3-regulares, VI

Desarrollos adicionales

Ideas:

- Fijamos los tres colores de un vértice concreto para evitar explorar subárboles que corresponden a permutar colores.

  (En general: identificamos simetrías y las usamos para evitar exploraciones innecesarias.)
- ¿Cómo sería la versión que nos da todas las soluciones?
- ¿Cómo tratar el problema cuando no suponemos 3-regularidad? Buscamos usar el mínimo de colores posible.
- Usando ideas similares, buscamos cómo plantear y resolver problemas de "vertex-coloring". Variante de optimización: usar, de nuevo, el mínimo de colores posible.
- ...

Algoritmos “greedy”, I

Como pronto veremos, ya conocemos ejemplos

Característica:

- La siguiente decisión es siempre “la que mejor parece” para el subproblema en curso;
- se toma esa decisión y nunca se reconsidera:

  no existe “backtracking”.
- Dado que, en ese momento, no se tiene perspectiva del problema global, esa decisión es arriesgada.
- Por tanto, se necesita una argumentación adicional, separada del algoritmo, que explique por qué es buena idea hacerlo así.

Algoritmos “greedy”, II

El principio “greedy” a argumentar en cada ocasión

“Greedy-choice property”:

Toda decisión válida que sea óptima para el subproblema local es también óptima para el problema global.

Algoritmos “greedy”, II

El principio “greedy” a argumentar en cada ocasión

“Greedy-choice property”:

Toda decisión válida que sea óptima para el subproblema local es también óptima para el problema global.

¿Cuándo hemos visto esto antes?

Algoritmos “greedy”, II

El principio “greedy” a argumentar en cada ocasión

“Greedy-choice property”:

Toda decisión válida que sea óptima para el subproblema local es también óptima para el problema global.

¿Cuándo hemos visto esto antes?

- Algoritmo de Dijkstra (single-source shortest paths),

Algoritmos “greedy”, II

El principio “greedy” a argumentar en cada ocasión

“Greedy-choice property”:

Toda decisión válida que sea óptima para el subproblema local es también óptima para el problema global.

¿Cuándo hemos visto esto antes?

- Algoritmo de Dijkstra (single-source shortest paths),
- algoritmo de Kruskal para encontrar árboles de expansión mínimos...

`https://en.wikipedia.org/wiki/Greedy_algorithm`.

Algoritmos “greedy”, III

O bien: Mochila, XVI

¿Obtendremos una solución óptima al aplicar el esquema “greedy” al problema de la mochila?

Algoritmos “greedy”, III

O bien: Mochila, XVI

¿Obtendremos una solución óptima al aplicar el esquema “greedy” al problema de la mochila?

- Objetos indivisibles: NO.

Algoritmos “greedy”, III

O bien: Mochila, XVI

¿Obtendremos una solución óptima al aplicar el esquema “greedy” al problema de la mochila?

- Objetos indivisibles: NO.
- Objetos divisibles: SÍ.
- A condición de poner un poco de cuidado en definir “decisión óptima”.

Algoritmos “greedy”, III

O bien: Mochila, XVI

¿Obtendremos una solución óptima al aplicar el esquema “greedy” al problema de la mochila?

- Objetos indivisibles: NO.
- Objetos divisibles: SÍ.
- A condición de poner un poco de cuidado en definir “decisión óptima”.
- Incluso para objetos indivisibles, nos da una información útil: una cota superior sobre el valor que se puede obtener a partir de un subproblema local.
- Si esa aproximación nos indica que un subproblema local no puede proporcionar una solución mejor que la mejor que se tiene hasta el momento, nos lo podemos ahorrar.

Algoritmos “greedy”, IV

O bien: Árboles de expansión, IV

`https://es.wikipedia.org/wiki/Algoritmo_de_Kruskal`

- Kruskal es un ejemplo clásico de algoritmo “greedy”.
- Proporciona un árbol de expansión minimal sólo al final: durante el proceso, se tienen fragmentos inconexos.
- Existe un algoritmo similar, pero en el cual siempre mantenemos un árbol de expansión conexo pero incompleto durante el proceso.
- ¿Lograrás diseñar este algoritmo por ti mism@? (NO busques en la Wikipedia “algoritmo de Prim” hasta haber completado tu propia solución.)

Giving change, I

O bien: Algoritmos "greedy", V

Dadas las denominaciones de determinadas monedas
(y provisión tan amplia de cada moneda como sea precisa),

- digamos, $d_1, \ldots, d_n$,
- y una cantidad concreta a alcanzar exactamente, M:
- ¿cómo lograrlo?

Relación con "la mochila" y con "Subset sum":

- Objetos "repetidos", podemos tomar cuantos queramos de cada tipo, y
- no hay pesos, pero la cantidad objetivo se ha de alcanzar exacta.

Giving change, II

O bien: Algoritmos "greedy", VI

Vamos tomando monedas de la denominación más elevada posible hasta alcanzar la cantidad deseada.

- Para muchas denominaciones, el algoritmo "greedy" funciona (es decir, da una solución con el mínimo número de monedas).
- Se llaman "canonical coin systems".
- Incluyen los casos típicos de la mayoría de los países:
  - 1, 2, 5, 10, 20, 50, 100, 200;
  - 1, 5, 10, 25, 50, 100;
  - 1, 29, 493;
  - ...
- Para otras denominaciones, no siempre!

`https://en.wikipedia.org/wiki/Change-making_problem`.

Giving change, III

O bien: Algoritmos "greedy", VI

- ¿Cómo se expresa el problema "giving change" en los términos de los esquemas de búsqueda combinatoria que hemos indicado anteriormente?
- Las denominaciones 1, 5, 10, 25 (las monedas de dólar de curso habitual) forman un "canonical coin system". Resuelve "giving change" en dólares mediante un algoritmo "greedy"
- Plantea y resuelve el mismo problema con las denominaciones del sistema euro, completo: Jutge P81629 en la lista Combinatorial Search Schemes (II).
- Encuentra casos en que el enfoque "greedy" no de la solución óptima.
- ¿Cómo obtener optimalidad en todos los casos?
  - "Backtracking" es siempre una opción.
  - Después del parcial veremos que será preferible Programación Dinámica.

Programación Dinámica, I

Lectura recomendada: `orígenes` por Richard Bellman en persona

Recordemos ("greedy-choice property"):

Toda decisión válida que sea óptima para el subproblema local es también óptima para el problema global.

Programación Dinámica, I

Lectura recomendada: `orígenes` por Richard Bellman en persona

Recordemos ("greedy-choice property"):

Toda decisión válida que sea óptima para el subproblema local es también óptima para el problema global.

Puntos clave hacia la Programación Dinámica:

- Es difícil, o imposible, argumentar cuál de las decisiones locales, óptima o no, llevará a la solución globalmente óptima:

  el mantra "greedy" es frecuentemente inaplicable;

Programación Dinámica, I

Lectura recomendada: `orígenes` por Richard Bellman en persona

Recordemos ("greedy-choice property"):

Toda decisión válida que sea óptima para el subproblema local es también óptima para el problema global.

Puntos clave hacia la Programación Dinámica:

- Es difícil, o imposible, argumentar cuál de las decisiones locales, óptima o no, llevará a la solución globalmente óptima:

  el mantra "greedy" es frecuentemente inaplicable;

- sin embargo, a veces, un primo suyo tal vez sea cierto, a saber, el Principio de Optimalidad de Bellman:

  la parte de la solución globalmente óptima que corresponde a cualquier subproblema local es, a su vez, una solución localmente óptima.

- Ejemplo: "Giving change".

Programación Dinámica, II

Modus operandi más habitual

Programación Dinámica tabulada:

- Organiza los subproblemas y sus soluciones óptimas en forma de tabla.
- Inventa y justifica una regla para llenar cada entrada de esa tabla,
- a partir de entradas de la tabla que sabes que puedes haber logrado llenar antes.
- Frecuentemente parece, a primera vista, ineficiente; pero no lo es tanto, y suele admitir además mejoras *ad-hoc*.
- (La Programación Dinámica no siempre es tabulada; pero en este curso sólo tratamos la variante tabulada.)

Giving change, IV

Cuando tus "monedas" no permiten solución "greedy"

En el ejemplo "giving change", primer planteamiento:

- Tabla T con tantas filas como denominaciones de moneda distintas, más una: "número de denominaciones en uso".

Giving change, IV

Cuando tus "monedas" no permiten solución "greedy"

En el ejemplo "giving change", primer planteamiento:

- Tabla T con tantas filas como denominaciones de moneda distintas, más una: "número de denominaciones en uso".
- Y con tantas columnas como indica la cantidad final M a obtener (o la más alta si se quiere poder usar la misma tabla para varios casos), más una.

Giving change, IV

Cuando tus "monedas" no permiten solución "greedy"

En el ejemplo "giving change", primer planteamiento:

- Tabla T con tantas filas como denominaciones de moneda distintas, más una: "número de denominaciones en uso".
- Y con tantas columnas como indica la cantidad final M a obtener (o la más alta si se quiere poder usar la misma tabla para varios casos), más una.
- La entrada $T[i, h]$ indica cuántas monedas se usan para obtener la cantidad $h \geq 0$, pero usando solamente las $i \geq 0$ denominaciones de moneda más pequeñas.

Giving change, IV

Cuando tus "monedas" no permiten solución "greedy"

En el ejemplo "giving change", primer planteamiento:

- Tabla T con tantas filas como denominaciones de moneda distintas, más una: "número de denominaciones en uso".
- Y con tantas columnas como indica la cantidad final M a obtener (o la más alta si se quiere poder usar la misma tabla para varios casos), más una.
- La entrada $T[i, h]$ indica cuántas monedas se usan para obtener la cantidad $h \geq 0$, pero usando solamente las $i \geq 0$ denominaciones de moneda más pequeñas.

$$T[i, h] =$$

Giving change, IV

Cuando tus "monedas" no permiten solución "greedy"

En el ejemplo "giving change", primer planteamiento:

- Tabla T con tantas filas como denominaciones de moneda distintas, más una: "número de denominaciones en uso".
- Y con tantas columnas como indica la cantidad final M a obtener (o la más alta si se quiere poder usar la misma tabla para varios casos), más una.
- La entrada $T[i, h]$ indica cuántas monedas se usan para obtener la cantidad $h \geq 0$, pero usando solamente las $i \geq 0$ denominaciones de moneda más pequeñas.

$$T[i, h] = \min(\ T[i-1, h],\ 1 + T[i, h - d_i]\ ).$$

Giving change, IV

Cuando tus "monedas" no permiten solución "greedy"

En el ejemplo "giving change", primer planteamiento:

- Tabla T con tantas filas como denominaciones de moneda distintas, más una: "número de denominaciones en uso".
- Y con tantas columnas como indica la cantidad final M a obtener (o la más alta si se quiere poder usar la misma tabla para varios casos), más una.
- La entrada $T[i, h]$ indica cuántas monedas se usan para obtener la cantidad $h \geq 0$, pero usando solamente las $i \geq 0$ denominaciones de moneda más pequeñas.

$$T[i, h] = \min(\ T[i-1, h],\ 1 + T[i, h - d_i]\ ).$$

si $i > 0$ y $h \geq d_i$.

Giving change, V

Muchos detalles a los que prestar atención

¿Significado preciso de las filas?

¡Afecta a la indexación de la lista de denominaciones!

- ¿$T[0, h]$? En particular $T[0, 0]$...

Giving change, V

Muchos detalles a los que prestar atención

¿Significado preciso de las filas?

¡Afecta a la indexación de la lista de denominaciones!

- ¿$T[0, h]$? En particular $T[0, 0]$...

  $T[0, 0] = 0$

Giving change, V

Muchos detalles a los que prestar atención

¿Significado preciso de las filas?

¡Afecta a la indexación de la lista de denominaciones!

- ¿$T[0, h]$? En particular $T[0, 0]$...

  $$T[0, 0] = 0$$

- $T[0, h]$ ha de indicar "imposible" para $h > 0$. ¿Qué haremos con esas entradas?

Giving change, V

Muchos detalles a los que prestar atención

¿Significado preciso de las filas?

¡Afecta a la indexación de la lista de denominaciones!

- ¿$T[0, h]$? En particular $T[0, 0]$...

  $T[0, 0] = 0$

- $T[0, h]$ ha de indicar "imposible" para $h > 0$. ¿Qué haremos con esas entradas?

  $T[0, h] =$ `float("inf")`

Giving change, V

Muchos detalles a los que prestar atención

¿Significado preciso de las filas?

¡Afecta a la indexación de la lista de denominaciones!

- ¿$T[0, h]$? En particular $T[0, 0]$...

  $T[0, 0] = 0$

- $T[0, h]$ ha de indicar "imposible" para $h > 0$. ¿Qué haremos con esas entradas?

  $T[0, h] =$ `float("inf")`

- ¿Dónde nos convienen las denominaciones?

Giving change, V

Muchos detalles a los que prestar atención

¿Significado preciso de las filas?

¡Afecta a la indexación de la lista de denominaciones!

- ¿$T[0, h]$? En particular $T[0, 0]$...

  $T[0, 0] = 0$

- $T[0, h]$ ha de indicar "imposible" para $h > 0$. ¿Qué haremos con esas entradas?

  $T[0, h] =$ `float("inf")`

- ¿Dónde nos convienen las denominaciones?

  En las posiciones 1 en adelante de una lista, `denoms`,

Giving change, V

Muchos detalles a los que prestar atención

¿Significado preciso de las filas?

¡Afecta a la indexación de la lista de denominaciones!

- ¿$T[0, h]$? En particular $T[0, 0]$...

  $T[0, 0] = 0$

- $T[0, h]$ ha de indicar "imposible" para $h > 0$. ¿Qué haremos con esas entradas?

  $T[0, h] =$ `float("inf")`

- ¿Dónde nos convienen las denominaciones?

  En las posiciones 1 en adelante de una lista, `denoms`,

  evitando d_0.

Giving change, V

Muchos detalles a los que prestar atención

¿Significado preciso de las filas?

¡Afecta a la indexación de la lista de denominaciones!

- ¿$T[0, h]$? En particular $T[0, 0]$...

  $T[0, 0] = 0$

- $T[0, h]$ ha de indicar "imposible" para $h > 0$. ¿Qué haremos con esas entradas?

  $T[0, h] =$ `float("inf")`

- ¿Dónde nos convienen las denominaciones?

  En las posiciones 1 en adelante de una lista, `denoms`,

  evitando d_0.

- Las necesitaremos ordenar.

Giving change, V

Muchos detalles a los que prestar atención

¿Significado preciso de las filas?

¡Afecta a la indexación de la lista de denominaciones!

- ¿$T[0, h]$? En particular $T[0, 0]$...

  $T[0, 0] = 0$

- $T[0, h]$ ha de indicar "imposible" para $h > 0$. ¿Qué haremos con esas entradas?

  $T[0, h] =$ `float("inf")`

- ¿Dónde nos convienen las denominaciones?

  En las posiciones 1 en adelante de una lista, `denoms`,

  evitando d_0.

- Las necesitaremos ordenar.

  ¿Qué "placeholder" podemos poner en `denoms[0]`?

Giving change, V

Muchos detalles a los que prestar atención

¿Significado preciso de las filas?

¡Afecta a la indexación de la lista de denominaciones!

- ¿$T[0, h]$? En particular $T[0, 0]$...

  $T[0, 0] = 0$

- $T[0, h]$ ha de indicar "imposible" para $h > 0$. ¿Qué haremos con esas entradas?

  $T[0, h] =$ `float("inf")`

- ¿Dónde nos convienen las denominaciones?

  En las posiciones 1 en adelante de una lista, `denoms`,

  evitando d_0.

- Las necesitaremos ordenar.

  ¿Qué "placeholder" podemos poner en `denoms[0]`?

  ```
  denoms[0] = float("-inf")
  ```

Giving change, VI

```
def gcdptable(denoms, upper_lim):
  t = {}   # t is a *dictionary* (!!) t[i,j] = t[(i,j)]
  for quantity in range(upper_lim + 1):
    "init table for no coins"
    t[0, quantity] = float("inf")
  t[0, 0] = 0
  for denom in range(1, len(denoms)): # Re-indexed 'denoms'
    for quantity in range(upper_lim + 1):
      if denoms[denom] <= quantity:
        "shall we use one more denoms[denom] coin?"
        t[denom, quantity] = min(
            t[denom - 1, quantity],
            1 + t[denom, quantity - denoms[denom]] )
      else:
        "cannot use that denomination anymore"
        t[denom, quantity] = t[denom - 1, quantity]
  return t
```

Giving change, VII

Pero, ¿cuál es realmente la solución completa?

Giving change, VII

Pero, ¿cuál es realmente la solución completa?

```
def trace(gctab, denoms, q):
  r = Counter() # Beware! 'denoms' are not re-indexed
  d = len(denoms) - 1
  while q:
    "non-generalizable: we can tell which case of the two"
    if d == 0:
      "only unit coins are used now"
      r[denoms[d]] += q
      break
    elif gctab[d, q] == gctab[d-1, q]:
      "coins of denoms[d] units were not employed"
      d -= 1
    else:
      r[denoms[d]] += 1
      q -= denoms[d]
  return r
```

Giving change, VIII

¿Realmente necesitamos toda la tabla?

¿Ha de estar siempre presente la denominación 1?

Condición necesaria y suficiente para poder resolver todos los casos.

- ¿Podemos simplificar la estructura de datos?
- Muchas veces, la manera de simplificar el programa es simplificar la estructura de datos.
- ¿Necesitamos tener siempre todas las filas?

Giving change, VIII

¿Realmente necesitamos toda la tabla?

¿Ha de estar siempre presente la denominación 1?

Condición necesaria y suficiente para poder resolver todos los casos.

- ¿Podemos simplificar la estructura de datos?
- Muchas veces, la manera de simplificar el programa es simplificar la estructura de datos.
- ¿Necesitamos tener siempre todas las filas?
- ¡En cada momento, nos basta tener la que estamos calculando!

Giving change, IX

```
dptable = [float("inf")]*(upper_lim + 1)
dptable[0] = 0
for i in range(1, upper_lim + 1):
   "dptable[i]: how many coins needed to add up to i"
   for coin in coins:
      "try using it"
      if coin <= i:
         dptable[i] = min(dptable[i], 1 + dptable[i-coin])
```

Calculando la tabla así, `dptable[h] > upper_lim` significa que no es posible alcanzar h, de lo cual informamos apropiadamente.

En otro caso, la solución está en `dptable[h]`.

Giving change, X

Para conservar la solución completa

Cada vez que se modifica la tabla principal, se anota el motivo del cambio en una tabla secundaria:

```
for coin in coins:
    if coin <= i:
        if 1 + dptable[i - coin] <= dptable[i]:
            dptable[i] = 1 + dptable[i - coin]
            best[i] = coin
```

¡Idea generalizable!

Giving change, XI

Para reconstruir la solución completa

Usando luego `best` (un `dict`), la reconstruimos así:

```
def trace(best, goal):
    coins = list()
    while goal:
        used = best[goal]
        coins.append(used)
        goal -= used
    return coins
```

Mochila, XVI

Brevísimamente

La mochila, ¿por Programación Dinámica?

Mochila, XVI

Brevísimamente

La mochila, ¿por Programación Dinámica?

- ¿Qué representa cada dimensión de la tabla?
- ¿Qué representa el contenido de las casillas?
- ¿Cómo queda entonces la correspondiente ecuación de Bellman?

Mochila, XVI

Brevísimamente

La mochila, ¿por Programación Dinámica?

- ¿Qué representa cada dimensión de la tabla?
- ¿Qué representa el contenido de las casillas?
- ¿Cómo queda entonces la correspondiente ecuación de Bellman?

$$T[i,h] = \max(\ T[i-1,h],\ v_i + T[i-1,h-w_i]\ )$$

. . . a condición de que. . .

Mochila, XVI

Brevísimamente

La mochila, ¿por Programación Dinámica?

- ¿Qué representa cada dimensión de la tabla?
- ¿Qué representa el contenido de las casillas?
- ¿Cómo queda entonces la correspondiente ecuación de Bellman?

$$T[i,h] = \max(\ T[i-1,h],\ v_i + T[i-1,h-w_i]\ )$$

. . . a condición de que. . .

$$w_i \leq h$$

Mochila, XVI

Brevísimamente

La mochila, ¿por Programación Dinámica?

- ¿Qué representa cada dimensión de la tabla?
- ¿Qué representa el contenido de las casillas?
- ¿Cómo queda entonces la correspondiente ecuación de Bellman?

$$T[i,h] = \max(\ T[i-1,h],\ v_i + T[i-1,h-w_i]\ )$$

. . . a condición de que. . .

$$w_i \leq h$$

A completar mediante cuidadosa consideración de las *boundary conditions*.

Supersecuencias, I

Conecta con aplicaciones en Bioinformática

Dadas dos secuencias (por ejemplo *strings*), ¿cuál es la secuencia más corta posible que tiene a ambas como subsecuencias?

(O una de ellas en caso de que haya varias.)

Supersecuencias, I

Conecta con aplicaciones en Bioinformática

Dadas dos secuencias (por ejemplo *strings*), ¿cuál es la secuencia más corta posible que tiene a ambas como subsecuencias?

(O una de ellas en caso de que haya varias.)

Como antes, empezamos por calcular solamente su longitud; añadimos luego código para trazar la supersecuencia solución.

Supersecuencias, I

Conecta con aplicaciones en Bioinformática

Dadas dos secuencias (por ejemplo *strings*), ¿cuál es la secuencia más corta posible que tiene a ambas como subsecuencias?

(O una de ellas en caso de que haya varias.)

Como antes, empezamos por calcular solamente su longitud; añadimos luego código para trazar la supersecuencia solución.

Si `r` es una de las supersecuencias comunes mínimas de `s` y `t`, ¿qué podemos averiguar sobre `r`?

Supersecuencias, II

Consideraciones

1. ¿Cómo serían los casos de secuencias de entrada vacías?

Supersecuencias, II

Consideraciones

1. ¿Cómo serían los casos de secuencias de entrada vacías?
2. ¿Cómo sería un caso en que ambas secuencias empiezan por la misma letra?
   - Entonces el resultado también.
   - Y el resto del resultado es...

Supersecuencias, II

Consideraciones

1. ¿Cómo serían los casos de secuencias de entrada vacías?
2. ¿Cómo sería un caso en que ambas secuencias empiezan por la misma letra?
   - Entonces el resultado también.
   - Y el resto del resultado es... un subproblema con secuencias más cortas.
3. Queda pendiente el caso de que las primeras letras de las dos secuencias sean diferentes.
   - Entonces el resultado ha de empezar por una de ellas.
   - Y el resto del resultado es...

Supersecuencias, II

Consideraciones

1. ¿Cómo serían los casos de secuencias de entrada vacías?
2. ¿Cómo sería un caso en que ambas secuencias empiezan por la misma letra?
   - Entonces el resultado también.
   - Y el resto del resultado es... un subproblema con secuencias más cortas.
3. Queda pendiente el caso de que las primeras letras de las dos secuencias sean diferentes.
   - Entonces el resultado ha de empezar por una de ellas.
   - Y el resto del resultado es... un subproblema con una secuencia mantenida igual y la otra más corta.

Supersecuencias, III

Tabla bidimensional

`S[i, j]`:

longitud de la supersecuencia más corta de `s[i:]` y `t[j:]`.

Supersecuencias, III

Tabla bidimensional

`S[i, j]`:

longitud de la supersecuencia más corta de `s[i:]` y `t[j:]`.

Si `s[i] == t[j]`:

Supersecuencias, III

Tabla bidimensional

`S[i, j]`:

longitud de la supersecuencia más corta de `s[i:]` y `t[j:]`.

Si `s[i] == t[j]`: `1 + S[i+1, j+1]`.

Si `s[i] != t[j]`:

Supersecuencias, III

Tabla bidimensional

`S[i, j]`:

longitud de la supersecuencia más corta de `s[i:]` y `t[j:]`.

Si `s[i] == t[j]`: `1 + S[i+1, j+1]`.

Si `s[i] != t[j]`: `1 + min(S[i+1, j], S[i, j+1])`.

Supersecuencias, III

Tabla bidimensional

`S[i, j]`:

longitud de la supersecuencia más corta de `s[i:]` y `t[j:]`.

Si `s[i] == t[j]`: `1 + S[i+1, j+1]`.

Si `s[i] != t[j]`: `1 + min(S[i+1, j], S[i, j+1])`.

Boundary conditions: cuando `s[i:]` y/o `t[j:]` es vacía.

Supersecuencias, III

Tabla bidimensional

`S[i, j]`:

longitud de la supersecuencia más corta de `s[i:]` y `t[j:]`.

Si `s[i] == t[j]`: `1 + S[i+1, j+1]`.

Si `s[i] != t[j]`: `1 + min(S[i+1, j], S[i, j+1])`.

Boundary conditions: cuando `s[i:]` y/o `t[j:]` es vacía.

Ojo! Esta vez, las entradas de la tabla dependen de otras entradas con índices superiores.

Supersecuencias, IV

Aproximación al programa

```
initialize
for i in reversed(range(len(s))):
    for j in reversed(range(len(t))):
        if s[i] == t[j]:
            S[i, j] = 1 + S[i+1, j+1]
        else:
            S[i, j] = 1 + min(S[i+1, j], S[i, j+1])
return S[0, 0]
```

Supersecuencias, V

Cómo trazar la solución

Método general: tabla secundaria que indica, para cada `i` y `j`, en caso de ser diferentes las letras `s[i]` y `t[j]`, si hemos de tomar en ese punto la letra de `s` o la de `t`.

Para construirla, en el caso `s[i] != t[j]` reemplazamos la operación `min` por una comprobación de desigualdad y apuntamos en la tabla secundaria de cuál de los dos strings dados viene la letra a usar (por ejemplo, con los valores -1 o 1, reservando el cero para cuando `s[i] == t[j]`).

Es también posible tomar la decisión mediante comparaciones del estilo de `S[i, j] == 1 + S[i+1, j]` sobre la tabla principal.

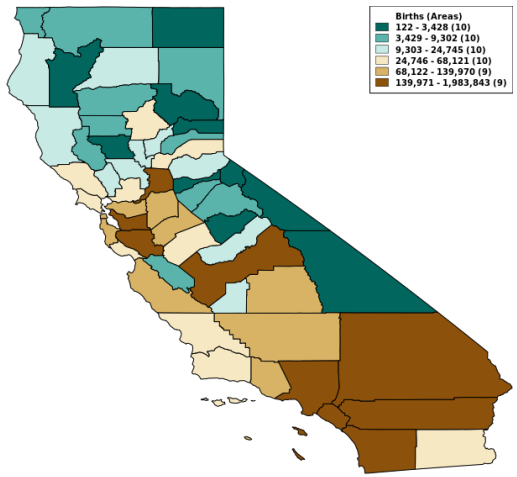
Discretización no supervisada

"Unsupervised Discretization" o "One-Dimensional Clustering"

Dada una lista de `float`'s, hay que particionarla en un número reducido de segmentos ("bins", "buckets", "clusters"...).

Resuelto (si que casi nadie se enterase) por los cartógrafos en la rama de cartografía llamada `choropleth maps`; la solución que describimos ahora, ellos la llaman `Jenks' natural breaks`.

Es un caso particular de la segmentación por "K-Means".



(Fuente: `Expert Health Data Programming, Inc (EHDP): Vitalnet`)

Segmentación por "K-Means"

"Clustering" que sigue el criterio de minimizar el error cuadrático

El caso general:

Con vectores de números reales en dimensión d.

- Datos: n vectores x_i, entero positivo k;
- Resultado: particionar los vectores en k clusters C_j;
- representaremos cada "cluster" C_j por un vector c_j (su centroide);
- los centroides han de minimizar el error cuadrático medio:

$$\frac{1}{n} \sum_j \sum_{x_i \in C_j} d(x_i, c_j)^2$$

Nota:

No exigimos que los c_j se escojan de entre los datos x_i.

Segmentación por "K-Means"

"Clustering" que sigue el criterio de minimizar el error cuadrático

El caso general:

Con vectores de números reales en dimensión d.

- Datos: n vectores x_i, entero positivo k;
- Resultado: particionar los vectores en k clusters C_j;
- representaremos cada "cluster" C_j por un vector c_j (su centroide);
- los centroides han de minimizar el error cuadrático medio:

$$\frac{1}{n} \sum_j \sum_{x_i \in C_j} d(x_i, c_j)^2$$

Nota:

No exigimos que los c_j se escojan de entre los datos x_i.

Malas noticias: *NP-hard* para dimensión 2 o más.

Y, ¿cómo lo resuelve la gente?

Si por milagro tuviéramos los centroides:

Entonces es fácil encontrar los "clusters": cada punto va a su centroide más próximo, porque, si no, el error crece.

Y, ¿cómo lo resuelve la gente?

Si por milagro tuviéramos los centroides:

Entonces es fácil encontrar los "clusters": cada punto va a su centroide más próximo, porque, si no, el error crece.

Si por milagro tuviéramos los "clusters":

Entonces es fácil encontrar los centroides: minimizamos $\sum_{x_i \in C} d(x_i, c)^2$ forzando la derivada a cero; el resultado (no podía ser otro) es que cada centroide queda en el baricentro de su "cluster" porque, si no, el error crece.

La heurística de Lloyd

Mucha gente la llama K-Means, confundiendo el problema con la solución aproximada

Vamos alternando

entre las dos cosas que sabemos hacer, empezando por k candidatos iniciales a centroide:

- recalcular los "clusters",
- recalcular los centroides,
- repetir.

`https://www.naftaliharris.com/blog/visualizing-k-means-clustering/`

K-Means: mínimo global en dimension 1, I

Dynamic Programming: estrategia de `Wang y Song`, a.k.a. `Jenks' Natural Breaks`

Input: número de "clusters" k, y n floats, con $n \geq k$; x_1 to x_n en orden creciente (pasa un `sort` si no).

K-Means: mínimo global en dimension 1, I

Dynamic Programming: estrategia de `Wang y Song`, a.k.a. `Jenks' Natural Breaks`

Input: número de "clusters" k, y n floats, con $n \geq k$; x_1 to x_n en orden creciente (pasa un `sort` si no).

Tabulamos: $C[i, m]$, coste de un "clustering" de los puntos x_1 to x_i en m "clusters", para $m \leq k$ y $m \leq i$; la solución está en $C[n, k]$.

K-Means: mínimo global en dimension 1, I

Dynamic Programming: estrategia de `Wang y Song`, a.k.a. `Jenks' Natural Breaks`

Input: número de "clusters" k, y n floats, con $n \geq k$; x_1 to x_n en orden creciente (pasa un `sort` si no).

Tabulamos: $C[i, m]$, coste de un "clustering" de los puntos x_1 to x_i en m "clusters", para $m \leq k$ y $m \leq i$; la solución está en $C[n, k]$.

Inicialización: $C[i, m] = 0$ si $m = 0$.

K-Means: mínimo global en dimension 1, I

Dynamic Programming: estrategia de `Wang y Song`, a.k.a. `Jenks' Natural Breaks`

Input: número de "clusters" k, y n floats, con $n \geq k$; x_1 to x_n en orden creciente (pasa un `sort` si no).

Tabulamos: $C[i, m]$, coste de un "clustering" de los puntos x_1 to x_i en m "clusters", para $m \leq k$ y $m \leq i$; la solución está en $C[n, k]$.

Inicialización: $C[i, m] = 0$ si $m = 0$.

¿Relación con "un cluster menos"? Identificamos el punto más pequeño del último "cluster".

K-Means: mínimo global en dimension 1, I

Dynamic Programming: estrategia de `Wang y Song`, a.k.a. `Jenks' Natural Breaks`

Input: número de "clusters" k, y n floats, con $n \geq k$; x_1 to x_n en orden creciente (pasa un `sort` si no).

Tabulamos: $C[i, m]$, coste de un "clustering" de los puntos x_1 to x_i en m "clusters", para $m \leq k$ y $m \leq i$; la solución está en $C[n, k]$.

Inicialización: $C[i, m] = 0$ si $m = 0$.

¿Relación con "un cluster menos"? Identificamos el punto más pequeño del último "cluster".

K-Means: mínimo global en dimension 1, I

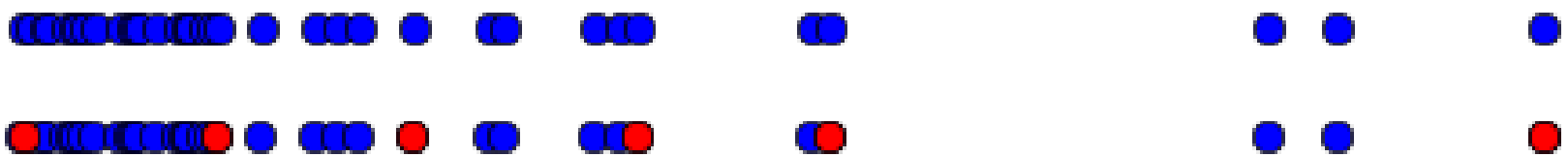
Dynamic Programming: estrategia de `Wang y Song`, a.k.a. `Jenks' Natural Breaks`

Input: número de "clusters" k, y n floats, con $n \geq k$; x_1 to x_n en orden creciente (pasa un `sort` si no).

Tabulamos: $C[i, m]$, coste de un "clustering" de los puntos x_1 to x_i en m "clusters", para $m \leq k$ y $m \leq i$; la solución está en $C[n, k]$.

Inicialización: $C[i, m] = 0$ si $m = 0$.

¿Relación con "un cluster menos"? Identificamos el punto más pequeño del último "cluster".



K-Means: mínimo global en dimension 1, II

¿Cuál es la diferencia entre m clusters y $m-1$ clusters?

$$C[i,m] = \min_{m \leq j \leq i}(C[j-1,m-1] + \sum_{j \leq \ell \leq i} d(x_\ell, c_{j,i})^2)$$

K-Means: mínimo global en dimension 1, II

¿Cuál es la diferencia entre m clusters y $m-1$ clusters?

$$C[i,m] = \min_{m \leq j \leq i}(C[j-1, m-1] + \sum_{j \leq \ell \leq i} d(x_\ell, c_{j,i})^2)$$

donde

$$c_{j,i} = \frac{1}{i-j+1} \sum_{j \leq \ell \leq i} x_\ell$$

K-Means: mínimo global en dimension 1, III

Demo available

`https://www.cs.upc.edu/~balqui/demoWSJ/`

Alpha stage!

- Algún día futuro me preocuparé por la estética,
- y por la usabilidad!...

Requiere:

- el número de clusters,
- los puntos que se supone que ya se han procesado y
- el nuevo punto a incorporar.

K-Means: mínimo global en dimension 1, IV

¿Cómo podemos hacerlo mejor?

Esta estrategia lleva a un algoritmo $O(n^3)$.

Mejora: no calcular individualmente cada $c_{j,i}$ sino actualizar $c_{j,i-1}$ (¿cómo hacerlo? Un poco de álgebra te lo dice.)

Ahorramos así una computación lineal que reduce el coste a $O(n^2)$.

(La alternativa de Jenks: in Cartografía sólo precisamos las fronteras entre clusters, sin los centroides. Es posible tunear la fórmula, reemplazando en el esquema de minimización los centroides por su definición.)

Hay quien afirma que se puede hacer en $O(n \log n)$. Ese texto no ha pasado revisión por pares.

Caminos mínimos, I

¿Aristas con costes? ¿Pueden ser negativos?

Problemas de caminos mínimos en grafos

Muy comunes: muchos problemas prácticos se pueden modelar así.

Empezamos por el caso "single-source": el vértice inicial de todos los caminos es fijo. El grafo es dirigido.

- ¿Costes variables en los arcos, o todos iguales?

  Si todos son iguales, ¡Breadth-First Search!
- ¿Hay arcos con costes negativos?
- Si no: Dijkstra (o extensiones como A*).
- Si los hay... ¿hay un ciclo de coste total negativo?

  En ese caso, el problema se complica. Solucionado muy recientemente.
- Si no los hay: Bellman-Ford:

Caminos mínimos, I

¿Aristas con costes? ¿Pueden ser negativos?

Problemas de caminos mínimos en grafos

Muy comunes: muchos problemas prácticos se pueden modelar así.

Empezamos por el caso "single-source": el vértice inicial de todos los caminos es fijo. El grafo es dirigido.

- ¿Costes variables en los arcos, o todos iguales?

  Si todos son iguales, ¡Breadth-First Search!
- ¿Hay arcos con costes negativos?
- Si no: Dijkstra (o extensiones como A*).
- Si los hay... ¿hay un ciclo de coste total negativo?

  En ese caso, el problema se complica. Solucionado muy recientemente.
- Si no los hay: Bellman-Ford:

  Un fragmento de un camino mínimo... ¡es mínimo!

Caminos mínimos, I

¿Aristas con costes? ¿Pueden ser negativos?

Problemas de caminos mínimos en grafos

Muy comunes: muchos problemas prácticos se pueden modelar así.

Empezamos por el caso "single-source": el vértice inicial de todos los caminos es fijo. El grafo es dirigido.

- ¿Costes variables en los arcos, o todos iguales?

  Si todos son iguales, ¡Breadth-First Search!
- ¿Hay arcos con costes negativos?
- Si no: Dijkstra (o extensiones como A*).
- Si los hay... ¿hay un ciclo de coste total negativo?

  En ese caso, el problema se complica. Solucionado muy recientemente.
- Si no los hay: Bellman-Ford:

  Un fragmento de un camino mínimo... ¡es mínimo!

  ¡Incluso en presencia de costes negativos!

Caminos mínimos, II

Si el Principio de Optimalidad se cumple...

Grafo dirigido de n vértices, vértice inicial s: "tabulamos" la distancia `dist[v, i]` de s a v en, como mucho, i pasos.

Si `dist[v, i]` es óptima y el último arco es (u, v), entonces `dist[u, i-1]` necesariamente es óptima.

$$dist[v, i] = \min(dist[v, i-1], dist[u, i-1] + cost[u, v])$$

Caminos mínimos, II

Si el Principio de Optimalidad se cumple...

Grafo dirigido de n vértices, vértice inicial s: "tabulamos" la distancia `dist[v, i]` de s a v en, como mucho, i pasos.

Si `dist[v, i]` es óptima y el último arco es (u, v), entonces `dist[u, i-1]` necesariamente es óptima.

$$dist[v, i] = \min(dist[v, i-1], dist[u, i-1] + cost[u, v])$$

```
for all v in V:
   dist[v] = float('inf')

dist[s] = 0
for i in range(1, n):
   for all the edges (u, v):
     if dist[v] > dist[u] + cost[u,v]:
       dist[v] = dist[u] + cost[u, v]
```

Caminos mínimos, II

Si el Principio de Optimalidad se cumple...

Grafo dirigido de n vértices, vértice inicial s: "tabulamos" la distancia `dist[v, i]` de s a v en, como mucho, i pasos.

Si `dist[v, i]` es óptima y el último arco es (u, v), entonces `dist[u, i-1]` necesariamente es óptima.

$$dist[v, i] = \min(dist[v, i-1], dist[u, i-1] + cost[u, v])$$

```
for all v in V:              # Bellman-Ford en PA2
   dist[v] = float('inf')
   prev[v] = None
dist[s] = 0
for i in range(1, n):
   for all the edges (u, v):
     if dist[v] > dist[u] + cost[u,v]:
       dist[v] = dist[u] + cost[u, v]
       prev[v] = u
```

Caminos mínimos, III

¿Qué es lo que decidimos considerar un subproblema?

"All-pairs shortest paths":

Dado un grafo (dirigido o no), ¿cuáles son las distancias más cortas entre todos los pares de vértices?

- Puede haber costes negativos, pero no ciclos de coste total negativo.
- Vértices de 0 a $N-1$,
- subproblemas definidos por un segmento inicial de esa secuencia de vértices;
- sólo se permiten los vértices de ese segmento inicial como vértices intermedios de un camino.
- Inicialmente: segmento nulo, no se permiten vértices como pasos intermedios; la distancia viene dada por los arcos individuales: si desde un vértice se alcanza directamente otro.

Caminos mínimos, IV

¡Recordemos comprobar el Principio de Optimalidad!

Esencia del algoritmo de `Floyd(-Warshall(-Roy))`:

- Si ya tenemos en nuestra tabla de distancias todas las que sólo usan vértices intermedios anteriores a k:

  ¿Cómo las usamos para contar también con k?

- La nueva opción k se usará o bien cero veces, ¡o bien exactamente una!

Caminos mínimos, IV

¡Recordemos comprobar el Principio de Optimalidad!

Esencia del algoritmo de `Floyd(-Warshall(-Roy))`:

- Si ya tenemos en nuestra tabla de distancias todas las que sólo usan vértices intermedios anteriores a k:

  ¿Cómo las usamos para contar también con k?

- La nueva opción k se usará o bien cero veces, ¡o bien exactamente una!

$$\begin{aligned}\text{dist(i, j, k)} = \min(\ &\text{dist(i, j, k-1)},\\ &\text{dist(i, k, k-1)} + \text{dist(k, j, k-1)}\ )\end{aligned}$$

Caminos mínimos, IV

¡Recordemos comprobar el Principio de Optimalidad!

Esencia del algoritmo de `Floyd(-Warshall(-Roy))`:

- Si ya tenemos en nuestra tabla de distancias todas las que sólo usan vértices intermedios anteriores a k:

  ¿Cómo las usamos para contar también con k?

- La nueva opción k se usará o bien cero veces, ¡o bien exactamente una!

$$\text{dist(i, j, k)} = \min(\ \text{dist(i, j, k-1)},\ \text{dist(i, k, k-1)} + \text{dist(k, j, k-1)}\ )$$

- Y si pasar por k es preferible, anotamos en la tabla secundaria que el mejor camino de i a j pasa por k.
- La tabla secundaria permite reconstruir recursivamente los caminos mínimos si se necesitan.

  (Existe una opción alternativa para esta reconstrucción: véase el enlace a Wikipedia dado arriba.)

Búsqueda exhaustiva, III

Si hay que probar todas las posibles soluciones, hagámoslo bien

¿No encuentras más opción que la búsqueda exhaustiva?

(No olvides preguntar si alguien ha demostrado NP-hardness; más explicaciones sobre esto en la segunda mitad del curso.)

1. Empieza por existencia, deja la optimización para después;
2. Usa la librería estándar para programar rápidamente una búsqueda exhaustiva, aunque sea exponencialmente lenta, y pruébala.
3. Se puede usar también para contabilizar repeticiones de subproblemas.
4. Si es demasiado lenta, plantea una solución por backtracking.
5. Subproblemas frecuentemente repetidos? Consideramos aplicar Programación Dinámica (dynamic programming), tal vez tras "backtracking", o tal vez directamente para empezar.

Búsqueda exhaustiva, IV

Una vez en este punto:

- Plantea el problema en términos de optimización.
- ¿"Best-first search"?

  (Es decir, A* y familia ("iterative deepening"...)

  `https://en.wikipedia.org/wiki/Best-first_search`.)
- ¿"Branch-and-bound"? ¿"Branch-and-cut"?, ¿AO* con "alpha-beta pruning"?...)