Programació i Algorítmica Avançades

Grau en Intel·ligència Artificial, FIB–UPC

José Luis Balcázar, Jordi Delgado

Dept. CS, UPC

2023–24, Quadrimestre de primavera

Contenido

Contenido

Algorítmica

En sentido laxo, incluye modelos abstractos de cálculo

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

Evaluación

Exámenes:

Parcial: 30% de la nota de curso.

Final: 50% de la nota de curso.

Práctica: 20% de la nota de curso.

Nota final: Máximo entre

- la nota del examen final y
- la nota de curso.

Lab

Y parte de los exámenes

He enviado a las direcciones de email `fib.upc` invitaciones al curso de `jutge.org` con el mismo nombre que nuestro curso. Quien

- no lo haya recibido o
- prefiera ser invitado con un email diferente,

que me lo avise por email (a `jose.luis.balcazar@upc.edu`) enviado desde la cuenta concreta que haya de recibir la invitación. Empezaremos a usar las listas de este curso en unas tres semanas, con una excepción: hoy.

Hoy

Repasamos recursividad y árboles

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

Contenido

Hacia una teoría de lenguajes formales

Maneras de procesar secuencias de símbolos

- Símbolo or letra: elementos de un conjunto finito fijo y no vacío, el alfabeto.
- Palabra: secuencia de letras, que solemos escribir yuxtapuestas (teniendo cuidado de que ello no introduzca ambigüedades).
- Las palabras tienen longitud; hay palabras finitas y también "palabras infinitas" (que no trataremos en este curso en absoluto).
- Ejemplo más común: cadenas de caracteres.
- Si Σ es un alfabeto, el conjunto de todas las palabras que podemos formar con sus letras se denota Σ^*.
- Los subconjuntos de Σ^* se llaman lenguajes formales.
- Cada lenguaje formal fijo plantea un problema decisional:

  Dada una palabra w, ¿pertenece o no a ese lenguaje formal?

Especificación de lenguajes formales, I

Enfoque decisional: Autómatas Finitos (Deterministas)

Un Autómata Finito Determinista sobre un alfabeto Σ consta de:

- Un conjunto finito de estados, de entre los cuales exactamente uno está designado como estado inicial (por tanto, el conjunto no es vacío).
- Un subconjunto de estados terminales (más propiamente denominados también estados aceptadores).
- Una función de transiciones válidas $(B, a) \rightarrow C$ (donde B y C son estados y $a \in \Sigma$) que indica el estado C (único, si lo hay) hacia el que existe una transición desde el estado B con el símbolo a.

Los símbolos del alfabeto se llaman frecuentemente terminales; pronto veremos la razón.

Especificación de lenguajes formales, II

Formalización y uso de los autómatas finitos

Si buscas "Autómata finito" en `Wikipedia`:

Formalmente, un autómata finito es una 5-tupla $(Q, \Sigma, q_0, \delta, F)$ donde:[6]

- Q es un conjunto finito de estados;
- Σ es un alfabeto finito;
- $q_0 \in Q$ es el estado inicial;
- $\delta: Q \times \Sigma \to Q$ es una función de transición;
- $F \subseteq Q$ es un conjunto de estados finales o de aceptación.

Normalmente recurrimos a explicar autómatas finitos mediante dibujos. Veamos un par de ejemplos:

- `un ejemplo`.
- `otro ejemplo`.

Representación gráfica de autómatas finitos

Mediante un grafo dirigido

- Un vértice por estado,
- un arco del estado B al C, etiquetado a, por cada transición $(B, a) \to C$.
- Marcas convencionales para el estado inicial y para los estados aceptadores.

Entonces, una palabra sobre el alfabeto puede trazar un camino sobre el grafo, empezando en el estado inicial.

Si el camino se puede completar y termina en un estado aceptador, el autómata acepta la palabra.

Especificación de lenguajes formales, III

Enfoque generativo: gramáticas regulares

Una gramática regular consta de:

- Alfabeto terminal Σ.
- Alfabeto no terminal Γ.
- Símbolo inicial $S \in \Gamma$.
- Reglas (a veces llamadas "producciones"): pares (B, a) o tripletas (B, a, C) donde $B \in \Gamma$, $C \in \Gamma$ y $a \in \Sigma$, siempre escritas $B \to a$ o $B \to aC$ respectivamente.

Especificación de lenguajes formales, IV

Un ejemplo de gramática regular

Terminales $\{a, b, c\}$; no terminales $\{S, A\}$:

$$\{ \quad S \to aS \quad S \to bA \quad S \to b \quad A \to cA \quad A \to c \quad \}$$

Derivaciones: reglas encadenadas que empiezan por el símbolo inicial y acaban en una palabra que sólo contiene terminales, tras recorrer cero o más palabras intermedias formadas por letras de ambos alfabetos.

Una gramática regular genera el lenguaje formal de las palabras que aparecen como última palabra de alguna derivación.

$$S \to aS \to aaS \to aabA \to aabcA \to aabccA \to aabccc$$

“Regular parsing”

El mismo problema, dado la vuelta

Análisis sintáctico regular: dada una gramática regular y una palabra,

> ¿hay una derivación de esa gramática que lleve a esa palabra?

Autómatas finitos indeterministas

Conexión entre gramáticas regulares y autómatas finitos

Dibujamos las gramáticas regulares también como grafos dirigidos.

- Los símbolos no terminales corresponden a vértices.
- Un vértice adicional "aceptador", con una marca convencional.
- Una marca convencional para el vértice que corresponde al símbolo inicial.
- A cada regla $B \to aC$ corresponde un arco dirigido del vértice B al C, etiquetado a.
- A cada regla $B \to a$ corresponde un arco dirigido del vértice B al vértice aceptador, etiquetado a.

Autómatas finitos indeterministas

Conexión entre gramáticas regulares y autómatas finitos

Dibujamos las gramáticas regulares también como grafos dirigidos.

- Los símbolos no terminales corresponden a vértices.
- Un vértice adicional "aceptador", con una marca convencional.
- Una marca convencional para el vértice que corresponde al símbolo inicial.
- A cada regla $B \to aC$ corresponde un arco dirigido del vértice B al C, etiquetado a.
- A cada regla $B \to a$ corresponde un arco dirigido del vértice B al vértice aceptador, etiquetado a.
- Ahora las derivaciones son caminos en el grafo, del vértice inicial al vértice aceptador.

¿A qué se parece? Y... ¿en qué se diferencia?

Especificación de lenguajes formales, V

O bien Autómatas finitos indeterministas, II

Un Autómata Finito Indeterminista sobre un alfabeto Σ consta de:

- Un conjunto de estados, de entre los cuales al menos uno está designado como estado inicial.
- Un subconjunto de estados terminales (más propiamente denominados también estados aceptadores).
- Una relación de transiciones válidas (B, a, C) (donde B y C son estados y $a \in \Sigma$) que indica estados C (¡puede haber varios!) hacia los que existe una transición desde el estado B con el símbolo a.

Entonces, una palabra sobre el alfabeto Σ puede trazar caminos sobre el grafo, empezando en un estado inicial.

Si algún camino se puede completar y termina en un estado aceptador, el autómata acepta la palabra.

Lenguajes regulares, I

Definiciones alternativas

Los siguientes enunciados son equivalentes:

1. L es el lenguaje generado por una gramática regular.
2. L es el lenguaje aceptado por un autómata finito indeterminista.
3. L es el lenguaje aceptado por un autómata finito determinista.

Los lenguajes formales L para los que se cumplen estas condiciones se llaman lenguajes regulares.

Podemos ver la equivalencia de las dos primeras a través del grafo que representa una gramática o un autómata.

La equivalencia con autómatas finitos deterministas requiere argumentación específica: la construcción "powerset".

“The Powerset”

Si L es el lenguaje aceptado por un autómata finito indeterminista, existe uno determinista que también lo acepta.

"The Powerset"

Si L es el lenguaje aceptado por un autómata finito indeterminista, existe uno determinista que también lo acepta.

Idea de la argumentación: seguimos "en paralelo", a la vez, todos los posibles caminos en el autómata indeterminista.

"The Powerset"

Si L es el lenguaje aceptado por un autómata finito indeterminista, existe uno determinista que también lo acepta.

Idea de la argumentación: seguimos "en paralelo", a la vez, todos los posibles caminos en el autómata indeterminista.

1. Creamos un estado para cada conjunto de estados del autómata indeterminista dado, N.
2. Llamamos estado inicial al formado por todos los estados iniciales del autómata indeterminista N.
3. Consideramos estados aceptadores todos los que contengan algún estado aceptador del autómata indeterminista N.
4. Definimos la función de transición desde el conjunto V con el símbolo a como el conjunto:

$$\{C \in N \mid (B, a, C) \text{ es una transición de } N \text{ y } B \in V\}$$

Si hay un camino aceptador en N, también lo hay en este nuevo autómata, que es determinista.

La “palabra vacía”

Una piedra en el zapato

En muchas argumentaciones sobre lenguajes formales, la palabra vacía requiere argumentación separada.

En estas breves horas introductorias, omitimos esas discusiones.

En un curso más detallado de lenguajes formales, se habría de tener en cuenta ese caso en cada argumentación y ajustar adecuadamente los enunciados.

Lenguajes regulares, II

Propiedades "positivas": lenguajes que son regulares

- Las uniones, intersecciones y complementarios de los lenguajes regulares son regulares.
- La concatenación de lenguajes regulares es regular:

$$LL' = \{ww' \mid w \in L,\ w' \in L'\}$$

- La estrella de Kleene de un lenguaje regular es regular:

$$L^* = \{w_0 \dots w_n \mid w_i \in L \text{ for all } i \leq n\}$$

(Para expresar concatenaciones y estrellas de Kleene, frecuentemente omitimos las "llaves" de conjunto.)

Lenguajes regulares, III

La propiedad de "bombeo", negativa: cómo ver que un lenguaje no es regular

Si L es un lenguaje regular, hay un número natural p tal que toda palabra $w \in L$ de longitud al menos p se puede descomponer en tres partes $w = xyz$ con las siguientes propiedades:

- $|y| > 0$ (la longitud de y es positiva, no nula),
- $|xy| \leq p$,
- $xy^*z \subseteq L$.

Razón: si la palabra w es aceptada por un autómata finito y es suficientemente larga, el camino en el grafo del autómata habrá de repetir estado.

Lenguajes regulares, III

La propiedad de "bombeo", negativa: cómo ver que un lenguaje no es regular

Si L es un lenguaje regular, hay un número natural p tal que toda palabra $w \in L$ de longitud al menos p se puede descomponer en tres partes $w = xyz$ con las siguientes propiedades:

- $|y| > 0$ (la longitud de y es positiva, no nula),
- $|xy| \leq p$,
- $xy^*z \subseteq L$.

Razón: si la palabra w es aceptada por un autómata finito y es suficientemente larga, el camino en el grafo del autómata habrá de repetir estado.

Consideramos

$$L = \{a^n b^n \mid n > 0\}.$$

Lenguajes regulares, III

La propiedad de "bombeo", negativa: cómo ver que un lenguaje no es regular

Si L es un lenguaje regular, hay un número natural p tal que toda palabra $w \in L$ de longitud al menos p se puede descomponer en tres partes $w = xyz$ con las siguientes propiedades:

- $|y| > 0$ (la longitud de y es positiva, no nula),
- $|xy| \leq p$,
- $xy^*z \subseteq L$.

Razón: si la palabra w es aceptada por un autómata finito y es suficientemente larga, el camino en el grafo del autómata habrá de repetir estado.

Consideramos

$$L = \{a^n b^n \mid n > 0\}.$$

¡NO es regular!

Especificación de lenguajes formales, VI

Un paso más allá de los lenguajes regulares: gramáticas incontextuales

$$\{ \quad S \to aSb \quad S \to ab \quad \}$$

Especificación de lenguajes formales, VI

Un paso más allá de los lenguajes regulares: gramáticas incontextuales

$$\{ \quad S \to aSb \quad S \to ab \quad \}$$

Otro ejemplo:

$$\{ \quad S \to () \quad S \to (S) \quad S \to SS \quad \}$$

(Lenguajes de Dyck: parentizaciones correctas con uno o más tipos de paréntesis... ¡cuidado con la palabra vacía!)

Especificación de lenguajes formales, VII

Añadimos una pila a los autómatas finitos

Autómatas indeterministas con pila: información adicional que se accede en el orden contrario al que se almacena.

Cada transición depende, además del estado y el símbolo, de lo que sacamos de la cima de la pila; y trae aparejados nuevos símbolos que entran en la pila (siempre por la cima).

En este caso, ya no son equivalentes la versión determinista y la versión indeterminista.

`http://lti.cs.vt.edu/FL/Books/VisFormalLang/html/PDA.html`

Lenguajes incontextuales, I

Nociones de muy amplia aplicación

Los siguientes enunciados son equivalentes:

1. L es el lenguaje generado por una gramática incontextual:

   son igual que las regulares, excepto que la parte derecha de cada regla puede ser cualquier secuencia de terminales y no terminales.

2. L es el lenguaje aceptado por un autómata con pila indeterminista.

Los lenguajes aceptados por autómatas finitos deterministas permiten algoritmos de "parsing" eficientes.

Muchas aplicaciones.

Lenguajes incontextuales, II

Casos no incontextuales

Existe una variante de la propiedad de "bombeo" apropiada para lenguajes incontextuales (no entramos en detalles).

Por ejemplo, esa propiedad indica que

$$\{a^n b^n c^n \mid n > 0\}.$$

no es incontextual.

Lenguajes incontextuales, II

Casos no incontextuales

Existe una variante de la propiedad de "bombeo" apropiada para lenguajes incontextuales (no entramos en detalles).

Por ejemplo, esa propiedad indica que

$$\{a^n b^n c^n \mid n > 0\}.$$

no es incontextual.

Existen diversos fenómenos no incontextuales en las lenguas humanas.

Especificación de lenguajes formales, VIII

Gramáticas sensibles al contexto

Un intento de modelar con gramáticas las lenguas humanas. . .

"Condicionamos" la aplicación de las reglas a su contexto: consideramos gramáticas con reglas de la forma

$$xAy \to xwy$$

que sería como "$A \to w$ pero sólo si el contexto son x e y".

Especificación de lenguajes formales, VIII

Gramáticas sensibles al contexto

Un intento de modelar con gramáticas las lenguas humanas. . .

"Condicionamos" la aplicación de las reglas a su contexto: consideramos gramáticas con reglas de la forma

$$xAy \to xwy$$

que sería como "$A \to w$ pero sólo si el contexto son x e y".

"Overkill": se pueden expresar lenguajes enormemente complejos.

Cualquier gramática en que las partes derechas de las reglas sean al menos tan largas como sus respectivas partes izquierdas ("gramáticas no contractivas") se puede transformar a esta definición.

Ejemplo:

$$\{ \quad S \to aSBc \quad S \to abc \quad cB \to Bc \qquad bB \to bb \qquad \}$$

La jerarquía de Chomsky

Con algunas salvedades referentes a la palabra vacía

Noam Chomsky, 1956:

Tipo 0: Gramáticas arbitrarias.

Corresponden a las máquinas de Turing y a los lenguajes recursivamente enumerables.

Tipo 1: Gramáticas sensibles al contexto; equivalentemente, gramáticas no contractivas.

Corresponden a las máquinas de Turing linealmente acotadas.

Tipo 2: Gramáticas incontextuales.

Corresponden a los autómatas indeterministas con pila y a los lenguajes incontextuales.

Tipo 3: Gramáticas regulares.

Corresponden a los autómatas finitos y a los lenguajes regulares.

Variantes

- Expresiones regulares.
- Expresiones regulares extendidas.
- Autómatas finitos con salida (transductores finitos, tratados en alguna otra asignatura).
- Autómatas finitos con pesos y/o probabilidades.
- Modelos de Markov (combinación de las dos ideas anteriores).
- Modelos de Markov ocultos (HMM, "hidden Markov models").
- "Mildly context sensitive languages".
- Gramáticas probabilistas.
- ...

Preguntas

Sobre temas aparentemente variados... que resultan ser el mismo

- El "tipo 0" de la jerarquía de Chomsky: ¿qué ocurre cuando consideramos gramáticas sin ningún tipo de restricciones?
- ¿Cambian mucho las cosas si un autómata con pila tiene, en vez de una sola pila, dos?
- ¿Con qué números reales estamos trabajando en realidad cuando desarrollamos todos los algoritmos del Cálculo Numérico?

Máquinas de Turing, I

Contexto

Estado del arte hacia 1935:

1. El "Entscheidungsproblem" de Gödel: ¿es posible avanzar en Matemáticas sin requerir ingenio?
2. Los enfoques de Gödel y de Church ya identifican la idea de cálculos que se efectúan sobre un dato, siguiendo un "proceso automático" o mecánico.
3. Ya está claro para ellos que la clave conceptual radica en la descomposición de un "cálculo" en "pasos".
4. Esos pasos han de ser "elementales", sin discusión posible (pero eso aún no se ha logrado).
5. Y también está claro que el tema es "importante" y tendrá consecuencias relevantes, porque permitirá argumentar sobre el Entscheidungsproblem y... quién sabe, más adelante, qué otras consecuencias. :)

Máquinas de Turing, II

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

Contribuciones de Turing mediante sus "máquinas"

1. Los pasos son elementales, sin discusión posible.

Máquinas de Turing, II

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

Contribuciones de Turing mediante sus "máquinas"

1. Los pasos son elementales, sin discusión posible.
2. Se pueden "componer funciones" combinando máquinas.

Máquinas de Turing, II

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

Contribuciones de Turing mediante sus "máquinas"

1. Los pasos son elementales, sin discusión posible.
2. Se pueden "componer funciones" combinando máquinas.
3. Se pueden concebir descripciones de todas las máquinas.

Máquinas de Turing, II

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

Contribuciones de Turing mediante sus "máquinas"

1. Los pasos son elementales, sin discusión posible.
2. Se pueden "componer funciones" combinando máquinas.
3. Se pueden concebir descripciones de todas las máquinas.
4. La descripción de una máquina puede proporcionarse como dato a otra máquina.

Máquinas de Turing, II

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

Contribuciones de Turing mediante sus "máquinas"

1. Los pasos son elementales, sin discusión posible.
2. Se pueden "componer funciones" combinando máquinas.
3. Se pueden concebir descripciones de todas las máquinas.
4. La descripción de una máquina puede proporcionarse como dato a otra máquina.
5. Podemos componer máquinas... ¡mediante otra máquina!

Máquinas de Turing, II

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

Contribuciones de Turing mediante sus "máquinas"

1. Los pasos son elementales, sin discusión posible.
2. Se pueden "componer funciones" combinando máquinas.
3. Se pueden concebir descripciones de todas las máquinas.
4. La descripción de una máquina puede proporcionarse como dato a otra máquina.
5. Podemos componer máquinas... ¡mediante otra máquina!
6. Y lo más importante y profundo, la máquina de Turing universal:

Máquinas de Turing, II

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

Contribuciones de Turing mediante sus "máquinas"

1. Los pasos son elementales, sin discusión posible.
2. Se pueden "componer funciones" combinando máquinas.
3. Se pueden concebir descripciones de todas las máquinas.
4. La descripción de una máquina puede proporcionarse como dato a otra máquina.
5. Podemos componer máquinas... ¡mediante otra máquina!
6. Y lo más importante y profundo, la máquina de Turing universal:
   - Recibe la descripción de una máquina M y la de un dato inicial para ella x;

Máquinas de Turing, II

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

Contribuciones de Turing mediante sus "máquinas"

1. Los pasos son elementales, sin discusión posible.
2. Se pueden "componer funciones" combinando máquinas.
3. Se pueden concebir descripciones de todas las máquinas.
4. La descripción de una máquina puede proporcionarse como dato a otra máquina.
5. Podemos componer máquinas... ¡mediante otra máquina!
6. Y lo más importante y profundo, la máquina de Turing universal:
   - Recibe la descripción de una máquina M y la de un dato inicial para ella x;
   - y simula la computación que haría M sobre x, obteniendo el mismo resultado.

Máquinas de Turing, III

"Demos" en la web

Se pueden ver como generalización de los autómatas finitos.

En vez del concepto original, veremos una simplificación propuesta por Emil Post sobre la versión original de Turing:

- Entrada y memoria (pila) se unifican en una cinta.
- Podemos movernos por ella en ambas direcciones: ya no hay que imponer uso como pila.
- Vale modificar los símbolos.
- Implementaciones online:

  `https://turingmachine.io/`

  `http://morphett.info/turing/turing.html`

  `https://turingmachinesimulator.com/`

Máquinas de Turing, IV

O bien: Especificación de lenguajes formales, IX

- Podemos cambiar como queramos el alfabeto (a condición de tener al menos dos símbolos, contando el "espacio vacío" como uno de ellos).
- Podemos poner en juego cualquier número de cintas.
- Podemos usar "cintas" multidimensionales.
- Podemos cambiar de muchas maneras el criterio para obtener el resultado o la decisión de aceptación.

Y, esencialmente, nada cambia.

Máquinas de Turing, V

Posibilidades

Y, además:

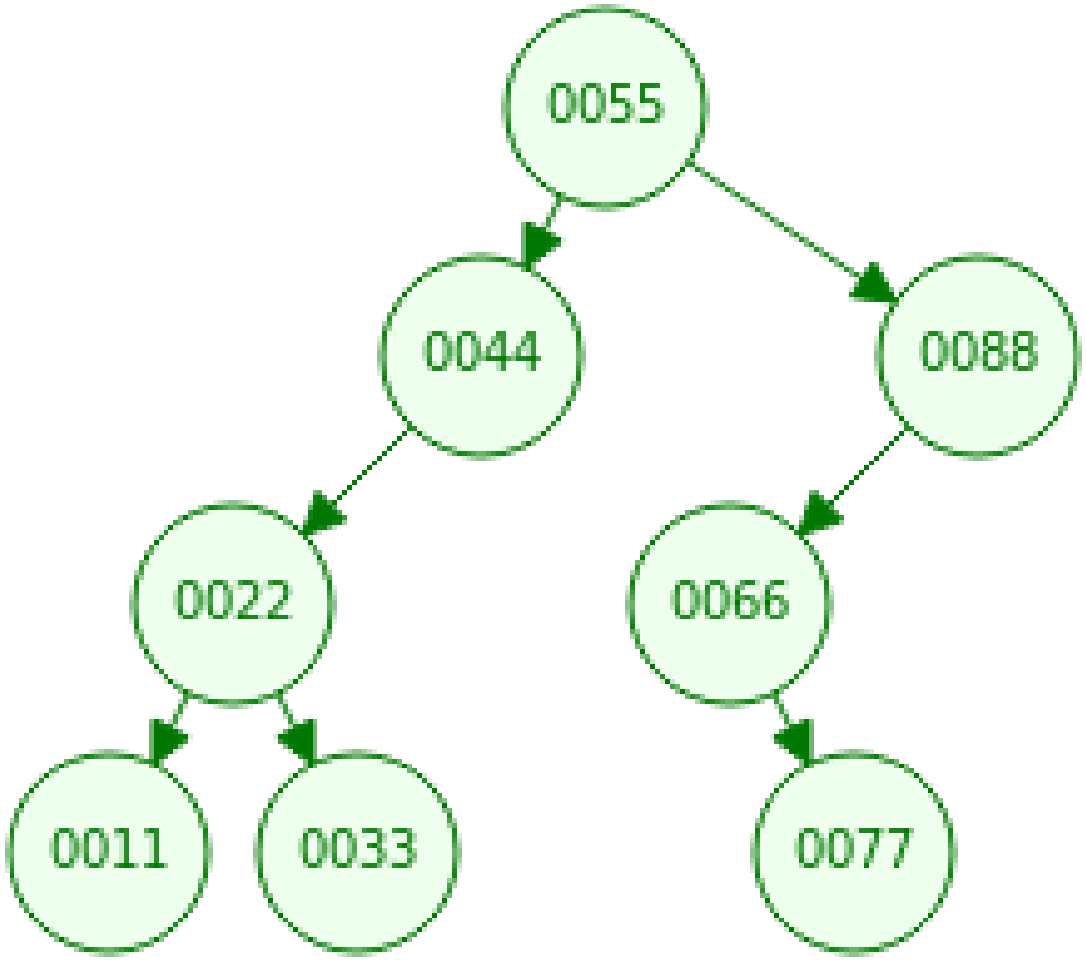
- Podemos simular derivaciones de gramáticas.
- Y viceversa: podemos describir mediante gramáticas los pasos de cálculo de una máquina de Turing.
- Podemos codificar datos, resultados y descripciones mediante números naturales.
- Y podemos usar números naturales para referirnos a máquinas o a sus datos.

Volveremos más adelante sobre las consecuencias de estos avances: el desarrollo de las teorías de la Calculabilidad y de la Complejidad Computacional.

Contenido

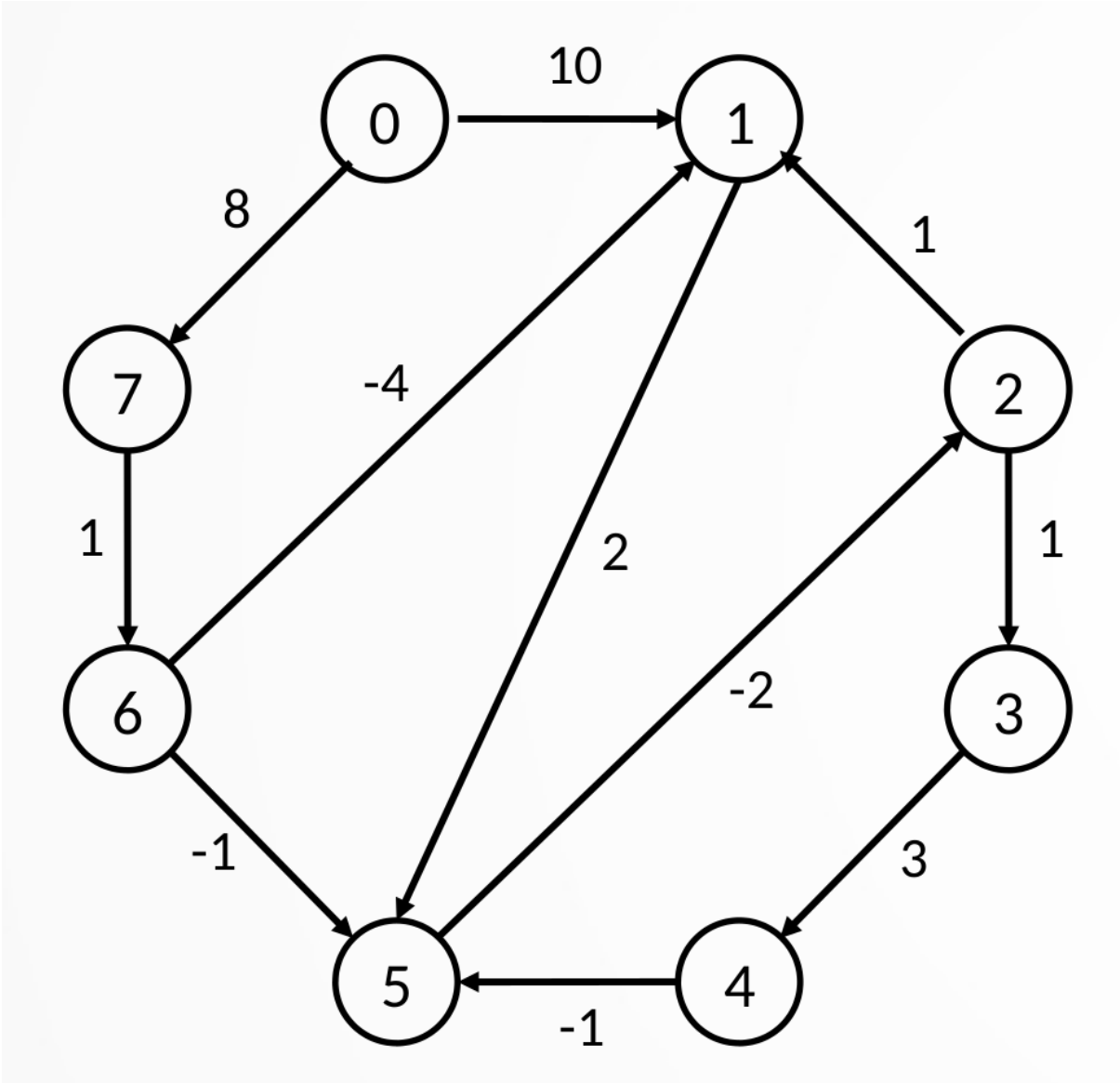
Recorridos de árboles

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.

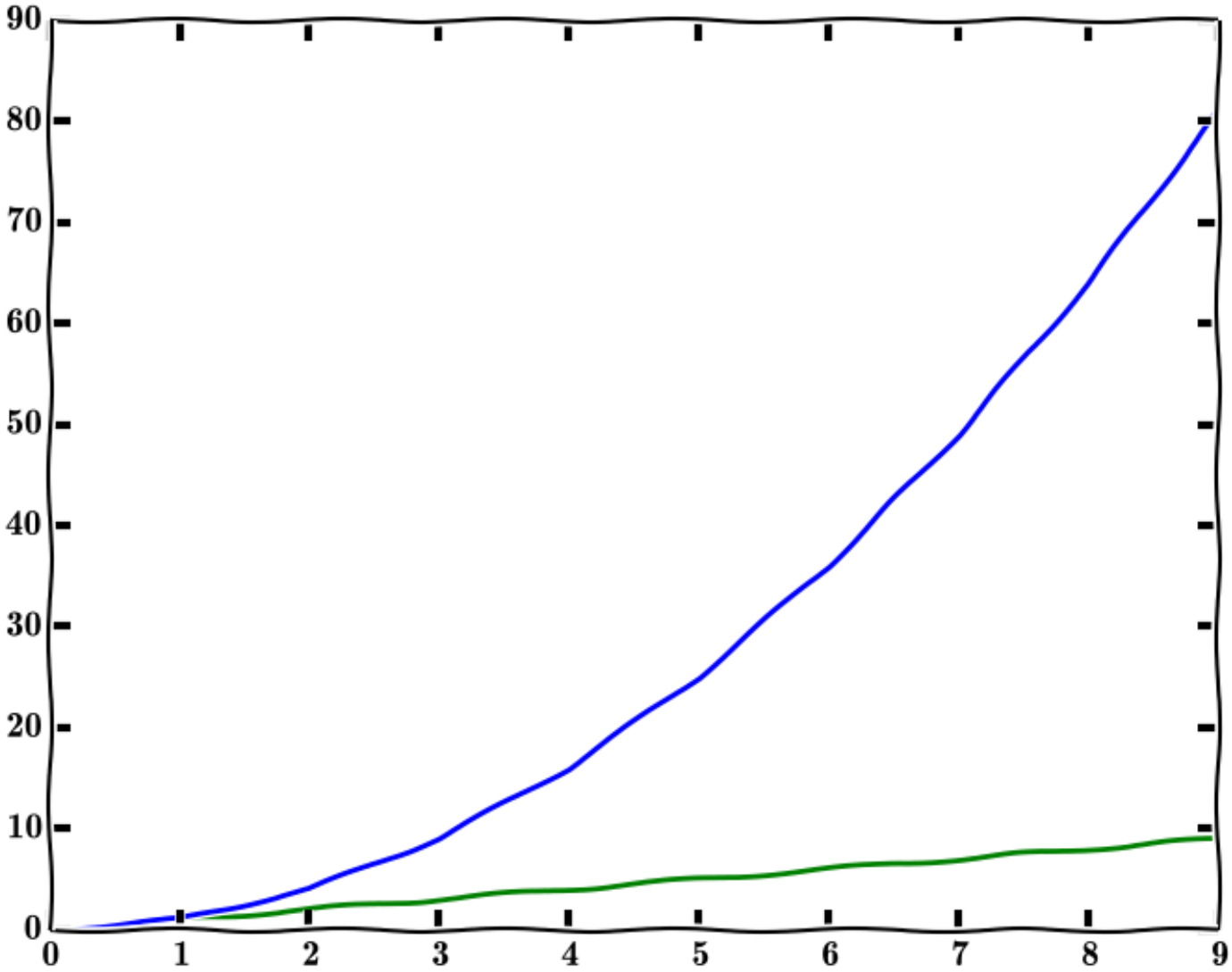
Búsqueda exhaustiva, II

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.

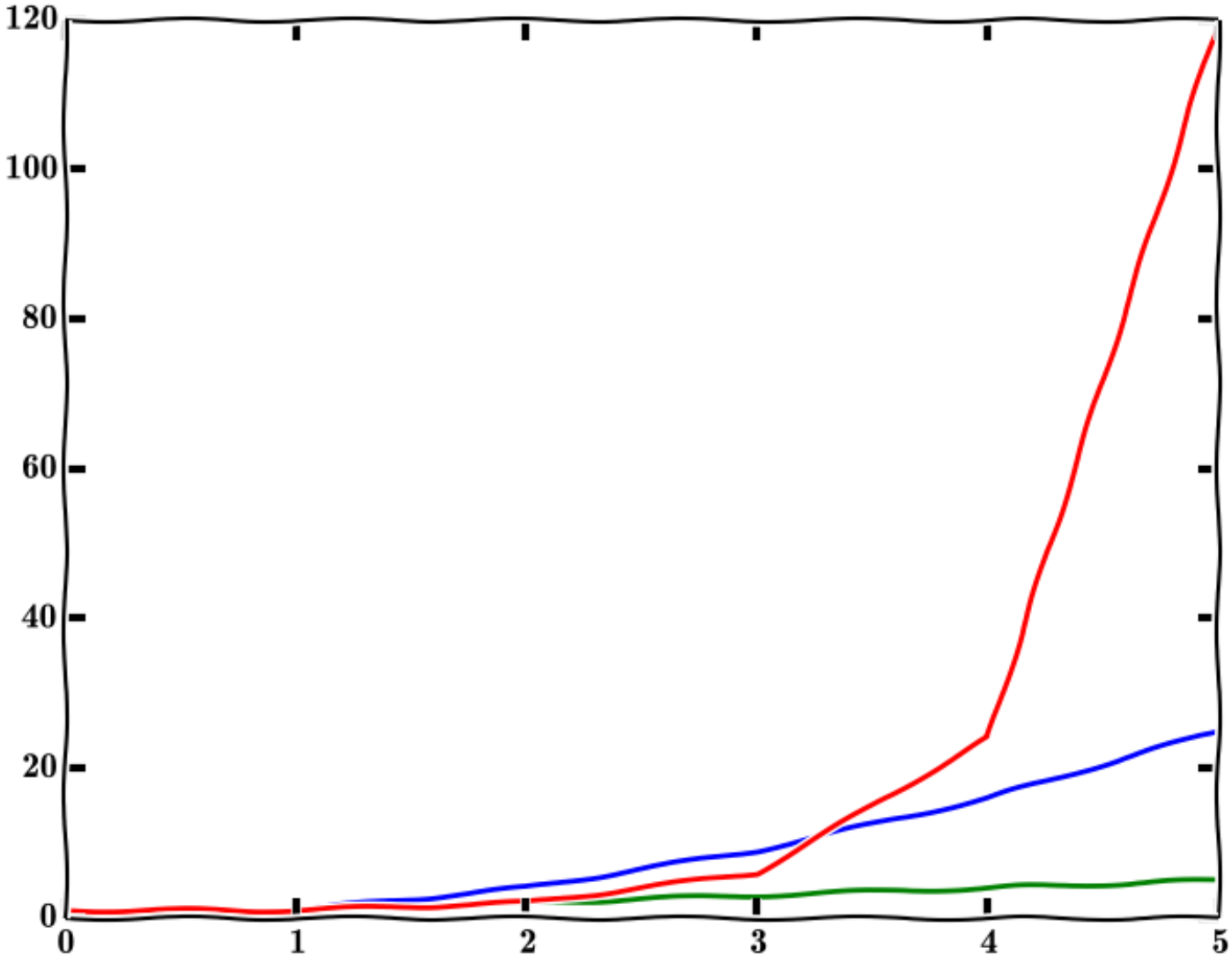
El factorial y el crecimiento exponential, I

No tomarás el nombre de la Exponencial en vano!



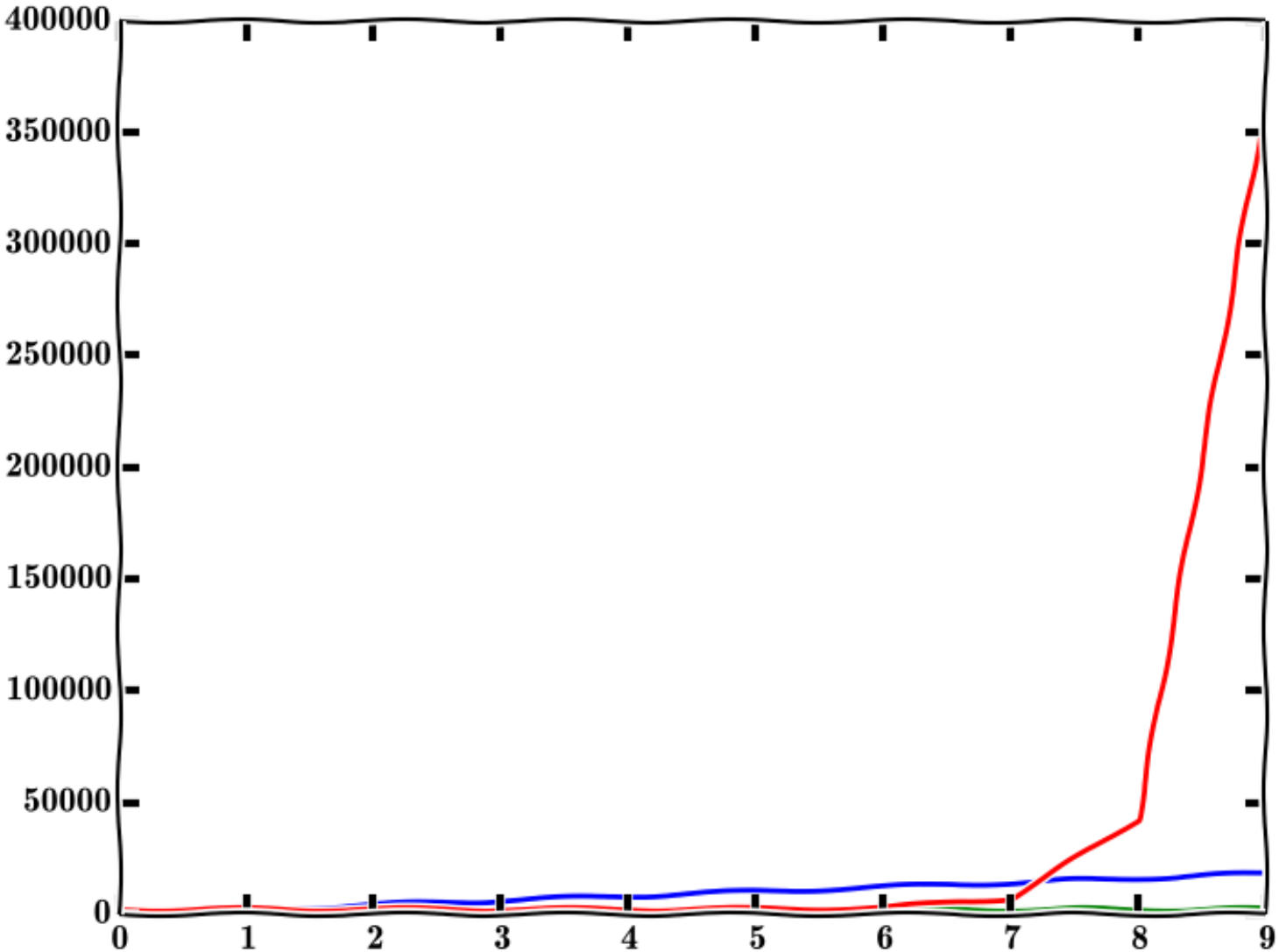
El factorial y el crecimiento exponential, I

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?

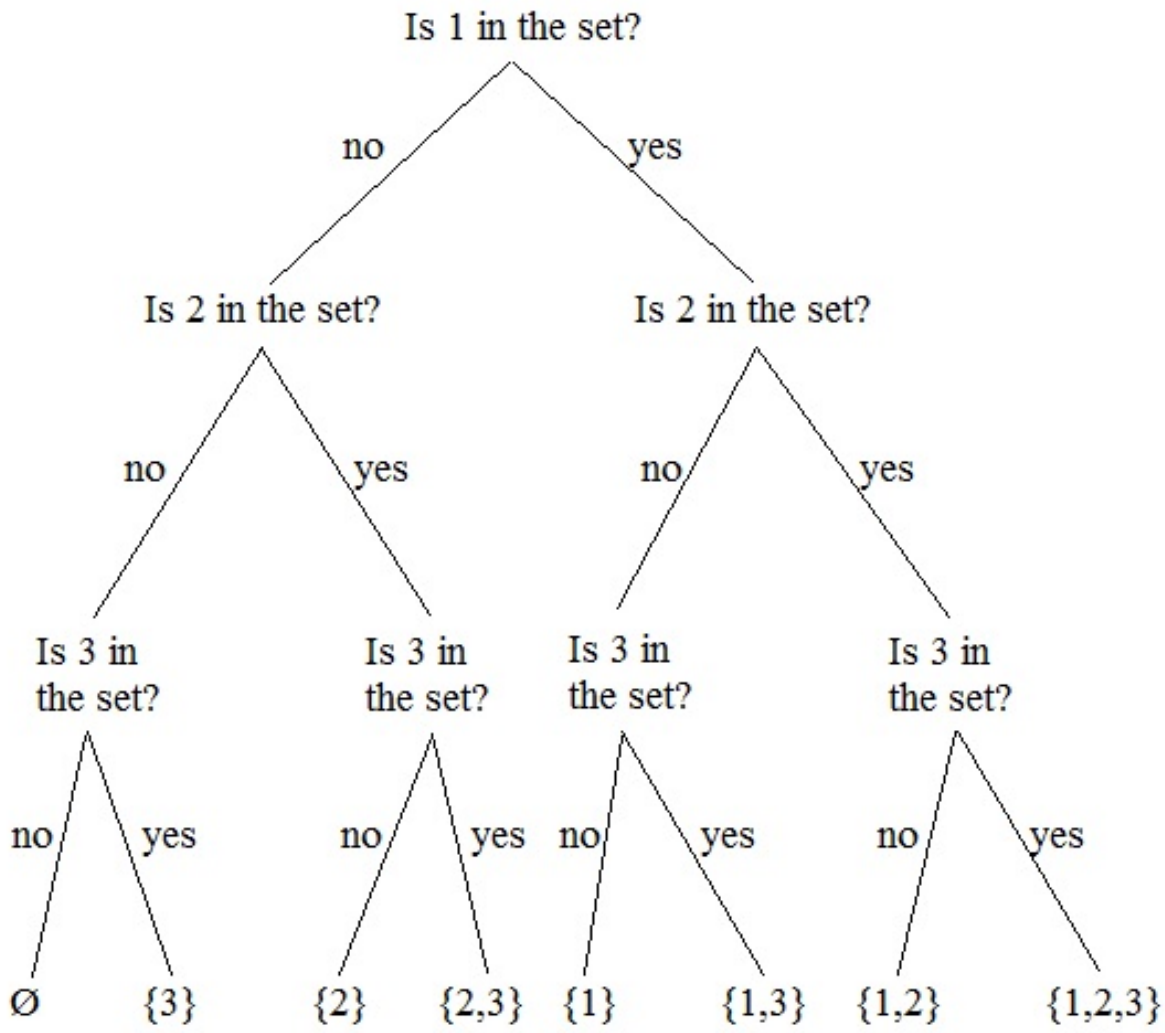
- ¿Conjuntos? "Set-based backtracking".
- ¿Permutaciones? "Permutation-based backtracking".
- ...

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)
```

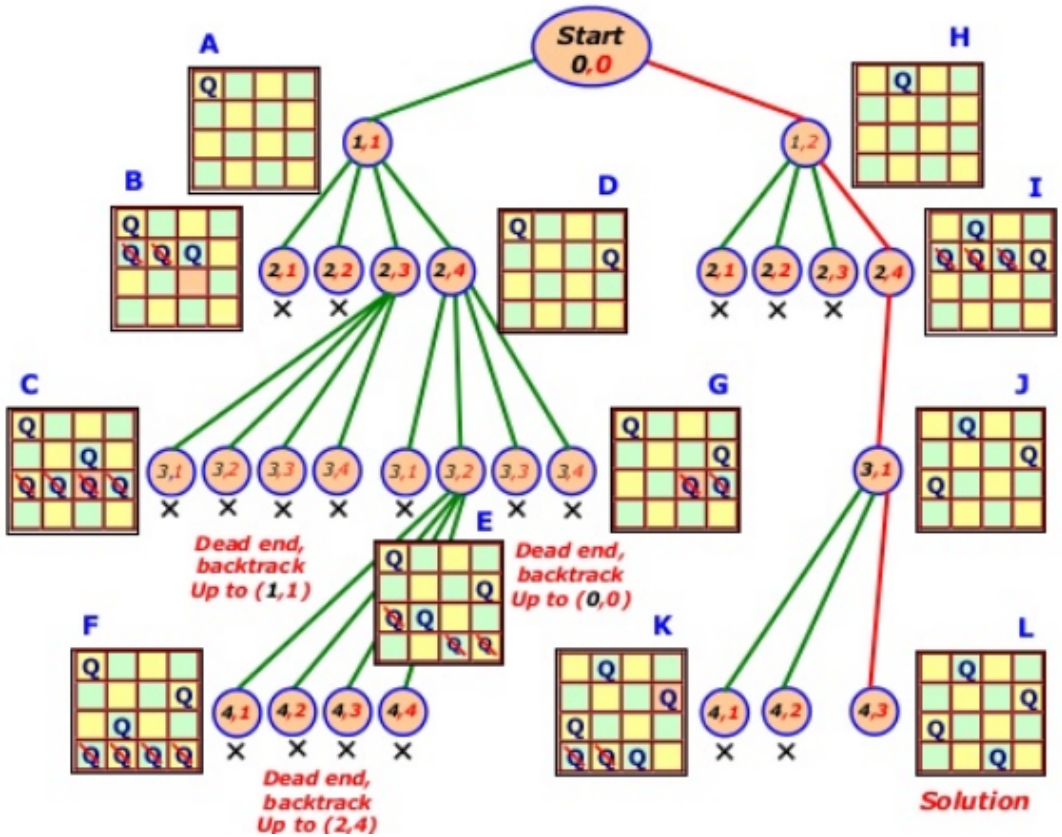
Mochila, XV

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:

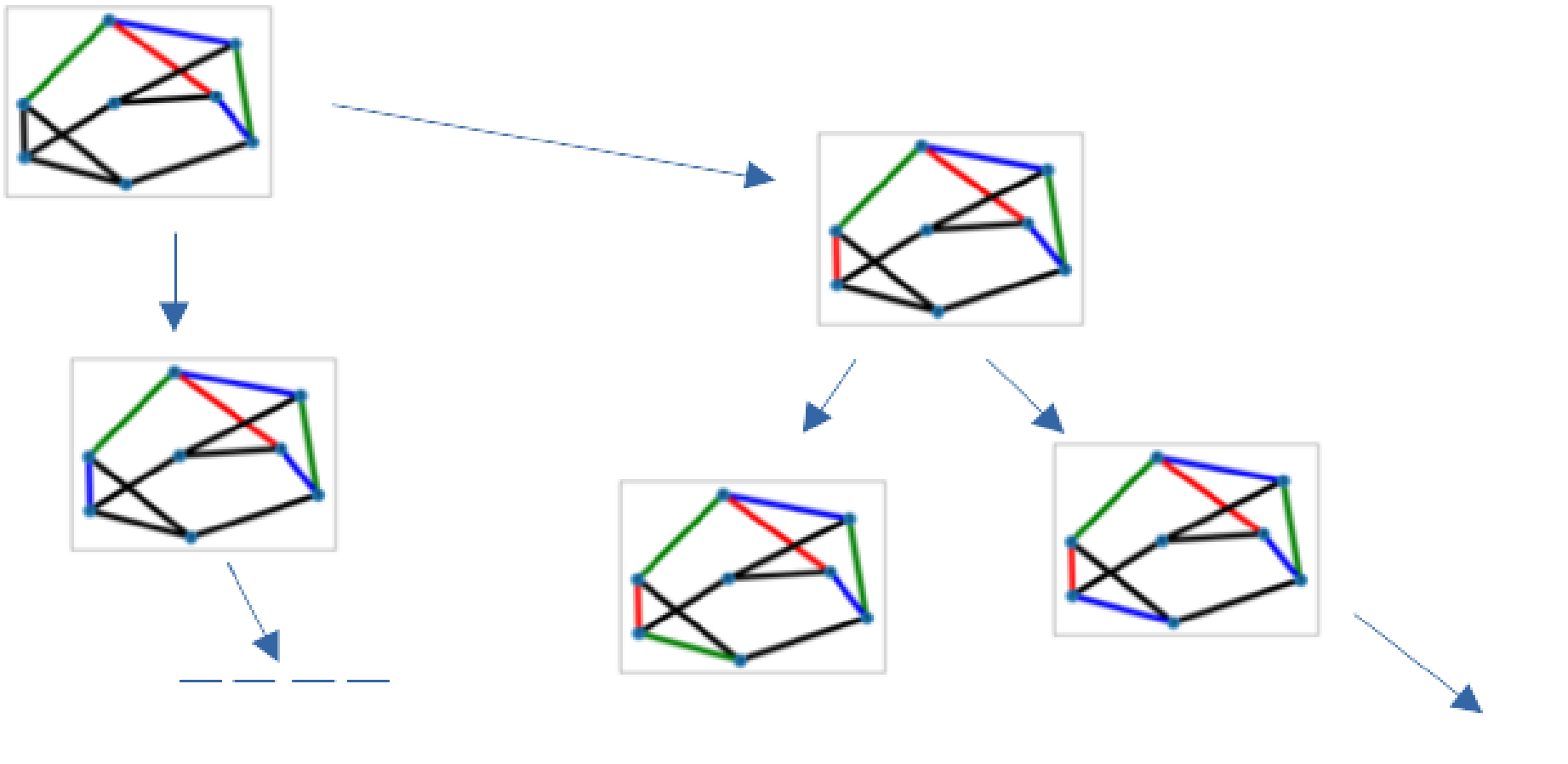
Dado un grafo 3-regular G, asígnense colores a las aristas usando tres colores de manera que en cada vértice haya una arista de cada color.

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 = {}
  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)):
    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()
  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 coinss:
    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
```

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.)

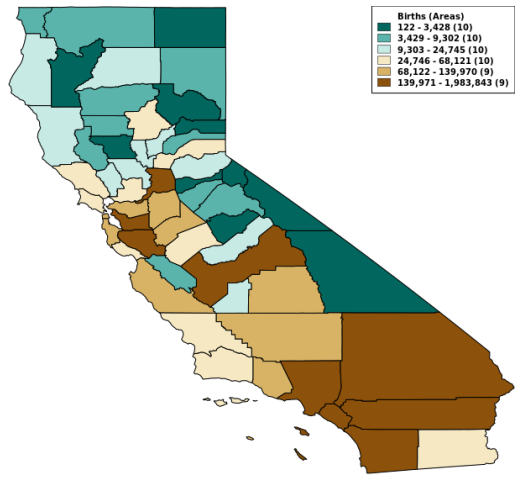
Discretización no supervisada

"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".



(Source: `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

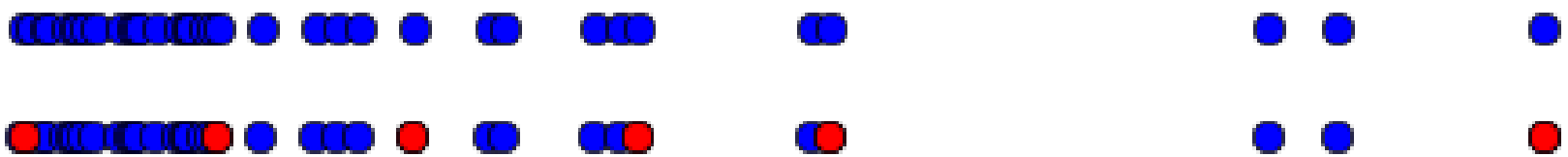
Dynamic Programming: estrategia de `Wang y Song`, a.k.a. `Jenks' Natural Breaks`

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?

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

K-Means: mínimo global en dimension 1, II

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

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

donde

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

K-Means: mínimo global en dimension 1, II

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

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

donde

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

$$h = m.$$

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.

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"?...)

Contenido

Preguntas

Sobre temas aparentemente variados... que resultan ser el mismo

- El "tipo 0" de la jerarquía de Chomsky: ¿qué ocurre cuando consideramos gramáticas sin ningún tipo de restricciones?
- ¿Cambian mucho las cosas si un autómata con pila tiene, en vez de una sola pila, dos?
- ¿Con qué números reales estamos trabajando en realidad cuando desarrollamos todos los algoritmos del Cálculo Numérico?
- Si partimos de unas pocas funciones muy sencillas y las vamos combinando por composición y mecanismos similares, ¿qué funciones obtenemos?
- Dejando aparte los lenguajes de programación "de propósito específico" orientados a una actividad concreta y considerando los "de propósito general", ¿hay problemas computacionales que podemos resolver en unos pero no en otros?

Decidibilidad

¿Existe un límite a lo que se puede hacer por software?

Varios enfoques heterogéneos...

¡que llevan todos al mismo punto!

Decidibilidad

¿Existe un límite a lo que se puede hacer por software?

Varios enfoques heterogéneos...

¡que llevan todos al mismo punto!

La pregunta de partida:

El "Entscheidungsproblem"
(`https://es.wikipedia.org/wiki/Entscheidungsproblem`):

- Precursores (Llull, Pascal, Leibniz, Babbage...)
- Hilbert y Ackermann, 1928, concretando preguntas anteriores: ¿qué algoritmo ("Entscheidungsverfahren") nos permite decidir si una fórmula de la lógica de primer orden es cierta?

Decidibilidad

¿Existe un límite a lo que se puede hacer por software?

Varios enfoques heterogéneos...

¡que llevan todos al mismo punto!

La pregunta de partida:

El "Entscheidungsproblem"
(`https://es.wikipedia.org/wiki/Entscheidungsproblem`):

- Precursores (Llull, Pascal, Leibniz, Babbage...)
- Hilbert y Ackermann, 1928, concretando preguntas anteriores: ¿qué algoritmo ("Entscheidungsverfahren") nos permite decidir si una fórmula de la lógica de primer orden es cierta?
- Turing, 1936: no lo hay; existen problemas indecidibles, y justo ése es uno de ellos.
- Independientemente, Gödel y Church, también en 1936, proponen nociones de indecidibilidad e incompletitud que, como iremos viendo, llevan a las mismas intuiciones.

Máquinas de Turing, II

Contexto

Estado del arte hacia 1935:

1. Los enfoques de Gödel y de Church ya identifican la idea de cálculos que se efectúan sobre un dato, siguiendo un "proceso automático" o mecánico.
2. También está ya claro que cada dato, a su vez, puede consistir de varios datos combinados en uno solo.
3. Y está claro que la clave conceptual radica en la descomposición de un cálculo en pasos; por ejemplo, mediante composición de funciones.
4. Los pasos han de ser elementales, sin discusión posible (esto no se logra aún en ese momento).
5. Pero también está claro que el tema es "importante" porque tendrá consecuencias importantes porque permitirá argumentar sobre el Entscheidungsproblem y... quién sabe más adelante qué otras consecuencias. :)

Dramatis Personae

Algunos nombres que van a irnos apareciendo

1. Las funciones recursivas parciales, que originariamente se definieron a través de diversas formas de recursividad... pero ya no, aunque conservan el nombre.
2. Las máquinas de Turing y, en particular, la Máquina de Turing Universal...
3. y su análogo, la función recursiva parcial universal.
4. Ambos métodos nos dan lugar a la noción de función calculable.
5. Un lenguaje será decidible si su función característica es calculable. La noción coincide con el Tipo 0 de Chomsky.
6. (Si se me escapa la expresión "lenguaje recursivo" quiero decir "lenguaje decidible".)

Dramatis Personae

Algunos nombres que van a irnos apareciendo

1. Las funciones recursivas parciales, que originariamente se definieron a través de diversas formas de recursividad... pero ya no, aunque conservan el nombre.
2. Las máquinas de Turing y, en particular, la Máquina de Turing Universal...
3. y su análogo, la función recursiva parcial universal.
4. Ambos métodos nos dan lugar a la noción de función calculable.
5. Un lenguaje será decidible si su función característica es calculable. La noción coincide con el Tipo 0 de Chomsky.
6. (Si se me escapa la expresión "lenguaje recursivo" quiero decir "lenguaje decidible".)
7. Y tendremos también "lenguajes indecidibles" y "funciones no calculables".

“Pairing” y “tupling”, I

Ejercicios previos

¿Conocemos el Hotel de Hilbert?

`https://en.wikipedia.org/wiki/Hilbert's_paradox_of_the_Grand_Hotel`

“Pairing” y “tupling”, I

Ejercicios previos

¿Conocemos el Hotel de Hilbert?

`https://en.wikipedia.org/wiki/Hilbert's_paradox_of_the_Grand_Hotel`

$$\langle x.y \rangle = (1/2)(x + y)(x + y + 1) + x + 1$$

Universalidad

Jugamos un ratito con el intérprete de Python.

Todos los tipos son, en el fondo, sólo uno:

Todo archivo que tengamos en la memoria de una máquina se puede ver:

- como un `string` probablemente muy largo: programas fuente, compilados, bases de datos...
- o bien como un número natural en binario.

Universalidad

Jugamos un ratito con el intérprete de Python.

Todos los tipos son, en el fondo, sólo uno:

Todo archivo que tengamos en la memoria de una máquina se puede ver:

- como un `string` probablemente muy largo: programas fuente, compilados, bases de datos...
- o bien como un número natural en binario.

Consideramos `strings` que describen expresiones.

- No usamos aún programas en toda su generalidad.
- Sólo componemos funciones, es decir, evaluamos expresiones.
- ¿Deja realmente la expresión de ser una `string`?

Sistemas Aceptables de Programación, I

Primero explicamos qué son; después, cómo obtenerlos

Todo "numerado": descripciones de algoritmos, datos y resultados.

Secuencia de funciones:

$\phi_0, \phi_1, \phi_2, \ldots, \phi_n, \ldots$

- Para todo $i \in \mathbb{N}$, $\phi_i : \mathbb{N} \to \mathbb{N}$ es la función calculada por el algoritmo descrito por el número i.
- Para $n \in \mathbb{N}$ (descripción de un dato), $\phi_i(n) \in \mathbb{N}$ es la descripción del resultado que obtiene el algoritmo descrito por el número i sobre el dato n.
- Son funciones parciales: el valor $\phi_i(n)$ puede no estar definido si i no describe un algoritmo correctamente, o si éste, sobre el dato n, nunca para. Notaciones: $\phi_i(n)\uparrow$, $\phi_i(n)\downarrow$.
- Requerimos algunas propiedades adicionales que explicamos más adelante.

Sistemas Aceptables de Programación, II

Las Máquinas de Turing proporcionan uno

Secuencia de funciones:

$\phi_0, \phi_1, \phi_2, \ldots, \phi_n, \ldots$

- $\phi_i : \mathbb{N} \to \mathbb{N}$ para todo $i \in \mathbb{N}$ es la función parcial calculada por la máquina de Turing cuya descripción es el número i.
- Un número concreto u describe la máquina universal: $\phi_u(i, n) = \phi_i(n)$.

Veremos otras maneras de construir estas enumeraciones; existen muchas, y resultan completamente equivalentes entre sí.

Sistemas Aceptables de Programación, II

Las Máquinas de Turing proporcionan uno

Secuencia de funciones:

$\phi_0, \phi_1, \phi_2, \ldots, \phi_n, \ldots$

- $\phi_i : \mathbb{N} \to \mathbb{N}$ para todo $i \in \mathbb{N}$ es la función parcial calculada por la máquina de Turing cuya descripción es el número i.
- Un número concreto u describe la máquina universal: $\phi_u(i, n) = \phi_i(n)$.

Veremos otras maneras de construir estas enumeraciones; existen muchas, y resultan completamente equivalentes entre sí.

Por cierto... $\phi_u(i, n)$ en realidad significa $\phi_u(\langle i.n \rangle)$.

Funciones Recursivas Parciales, I

De momento, entre los números naturales

La versión clásica de las funciones recursivas parciales es entre tuplas de números.

Se complican los razonamientos y la notación, porque hay que andar concretando tamaños de cada tupla.

Aquí simplificaremos.

- Empezamos considerando sólo funciones (posiblemente) parciales de $\mathbb{N} \to \mathbb{N}$.
- Tenemos ya cómo representar pares de números en uno solo:

$$\langle x.y \rangle = (1/2)(x+y)(x+y+1) + x + 1$$

- Si $n \neq 0$, entonces $n = \langle x.y \rangle$ representa el par $(x, y) \in \mathbb{N} \times \mathbb{N}$.

“Pairing” y “tupling”, II

Tuplas en sentido más amplio

¿Tuplas de 3 naturales en uno?

¡Fácil!

$$\langle x.y.x \rangle = \langle x.\langle y.z \rangle \rangle$$

- Generaliza a cualquier longitud fija.
- Sobrecargamos el símbolo.

“Pairing” y “tupling”, II

Tuplas en sentido más amplio

¿Tuplas de 3 naturales en uno?

¡Fácil!

$$\langle x.y.x\rangle = \langle x.\langle y.z\rangle\rangle$$

- Generaliza a cualquier longitud fija.
- Sobrecargamos el símbolo.

Pero... ¿tuplas de longitud variable?

Se complica una pizca, pero poco:

- $\langle x\rangle = \langle x.0\rangle$,
- $\langle x, y\rangle = \langle x.\langle y\rangle\rangle = \langle x.\langle y.0\rangle\rangle$ y, en general,
- para $m > 1$, $\langle x_0, \ldots, x_{m-1}\rangle = \langle x_0.\langle x_1, \ldots, x_{m-1}\rangle\rangle$.

“Pairing” y “tupling”, II

Tuplas en sentido más amplio

¿Tuplas de 3 naturales en uno?

¡Fácil!

$$\langle x.y.x\rangle = \langle x.\langle y.z\rangle\rangle$$

- Generaliza a cualquier longitud fija.
- Sobrecargamos el símbolo.

Pero... ¿tuplas de longitud variable?

Se complica una pizca, pero poco:

- $\langle x\rangle = \langle x.0\rangle$,
- $\langle x, y\rangle = \langle x.\langle y\rangle\rangle = \langle x.\langle y.0\rangle\rangle$ y, en general,
- para $m > 1$, $\langle x_0, \ldots, x_{m-1}\rangle = \langle x_0.\langle x_1, \ldots, x_{m-1}\rangle\rangle$.
- Naturalmente, el 0 codifica, según este esquema, la secuencia vacía $\langle\rangle$.

“Pairing” y “tupling”, III

Proyecciones del par

Funciones inversas para el “tupling”.

Funciones de proyección, π^L y π^R:

- permiten descodificar un par,
- seleccionando individualmente cada una de las componentes: $\langle \pi^L(x).\pi^R(x) \rangle = x$ si $x > 0$.
- Sobreyectivas.
- Existen funciones similares para tamaños mayores de tupla.
- Convenimos que $\pi^L(0) = 0$ y $\pi^R(0) = 0$.

“Pairing” y “tupling”, IV

Tupla sufijo

Función de tupla sufijo:

Denotada $\pi^*(\langle w.k\rangle)$ donde w es una tupla de al menos k componentes, devuelve la tupla que consta de las componentes de w de la k-ésima en adelante.

Como ejemplo, sea

$$w = \langle x, y, z\rangle = \langle x.\langle y, z\rangle\rangle = \langle x.\langle y.\langle z.\langle\rangle\rangle\rangle\rangle = \langle x.\langle y.\langle z.0\rangle\rangle\rangle:$$

$$\pi^*(\langle\langle x, y, z\rangle.0\rangle) = \langle x, y, z\rangle,$$
$$\pi^*(\langle\langle x, y, z\rangle.1\rangle) = \langle y, z\rangle,$$
$$\pi^*(\langle\langle x, y, z\rangle.2\rangle) = \langle z\rangle = \langle z.0\rangle,$$
$$\pi^*(\langle\langle x, y, z\rangle.3\rangle) = \langle\rangle = 0.$$

Definición:

- $\pi^*(0) = 0$,
- $\pi^*(\langle w.0\rangle) = w$,
- para $k > 0$, $\pi^*(\langle w.k\rangle) = \pi^*(\langle \pi^R(w).k - 1\rangle)$

“Pairing” y “tupling”, V

Proyección general

Función de proyección:

Denotada $\pi(\langle w.k \rangle)$ donde w es una tupla de al menos k componentes, devuelve la componente k de w.
Por ejemplo:

$$\pi(\langle \langle x, y, z \rangle .1 \rangle) = y.$$

$$\pi(\langle \langle x, y, z \rangle .2 \rangle) = z.$$

Definición:

- $\pi(0) = 0$,
- $\pi(\langle x.k \rangle) = \pi^L(\pi^*(\langle x.k \rangle))$.

Funciones Recursivas Parciales, II

Punto de partida para construir funciones

Funciones básicas:

- la identidad,
- la función constante 1, $k_1(x) = 1$,
- la suma $\langle x.y \rangle \to x + y$,
- el producto $\langle x.y \rangle \to x * y$,
- la diferencia modificada $\langle x.y \rangle \to x \dot{-} y = \max(x - y, 0)$,
- la función de tupla sufijo π^*,
- y la función general de proyección π.

Funciones Recursivas Parciales, III

Primera aproximación

Operativa:

- Composición: $\phi \circ \psi(n) = \phi(\psi(n))$.
- Formación de pares ("pairing"): $\langle \phi.\psi \rangle(n) = \langle \phi(n).\psi(n) \rangle$.
- Minimización sobre una función ϕ, definida por

  $\psi(x) = \min\{z \mid \phi(\langle x.z \rangle) = 1 \wedge \forall w < z(\phi(\langle x.w \rangle)\downarrow)\}$

  si este conjunto es no vacío, y $\psi(x)\uparrow$ en otro caso.

  Procuramos aplicar este operador sobre funciones totales para poder simplificar la última cláusula.

Las funciones recursivas parciales son las que podemos construir a partir de las básicas, combinándolas con estas tres operaciones.

"Minimización" es el nombre clásico; la denotamos $\psi = \mu\phi$. El nombre moderno debería ser "búsqueda lineal".

Funciones Recursivas Parciales, IV

¿Por qué precisamos funciones parciales?

Abreviamos $x \in \mathrm{Dom}\ \phi$ como $\phi(x)\downarrow$, y $x \notin \mathrm{Dom}\ \phi$ como $\phi(x)\uparrow$.

En la expresión

$$\psi(x) = \min\{z \mid \phi(\langle x.z\rangle) = 1 \wedge \forall w < z(\phi(\langle x.w\rangle)\downarrow)\},$$

incluso en el caso de que ϕ sea una función total y, por tanto, la cláusula "para todo w" siempre se cumple...

¡cabe la posibilidad de que el z buscado no exista!

En ese caso, $\psi(x)\uparrow$.

¿Y si pretendemos "completar" la función haciéndola valer, por ejemplo, 0 en caso de indefinición?

No hay problema en "definirla" así, pero, entonces, ¿la podremos realmente calcular? ("Spoiler": no.)

Funciones Recursivas Parciales, V

Esquemas adicionales

- Distinción de casos: si χ_A, f y g son recursivas totales, también lo es

$$h(x) = \begin{cases} f(x) & \text{si } x \in A \\ g(x) & \text{si } x \notin A. \end{cases}$$

- Cuantificación existencial acotada: si P es un predicado recursivo, también lo es:

$$P'(\langle x.t \rangle) = 1 \iff \exists y (y \leq t \wedge P(\langle x.y \rangle) = 1).$$

- Cuantificación universal acotada: si P es un predicado recursivo, también lo es:

$$P''(\langle x.t \rangle) = 1 \iff \forall y (y \leq t \Rightarrow P(\langle x.y \rangle) = 1).$$

(Llamamos "predicados" a funciones totales que sólo devuelven los valores 0 o 1.)

Funciones Recursivas Parciales, VI

Recursión primitiva

A partir de las funciones totales f y g, s se define por recursión primitiva si existe k tal que:

- $s(x) = g(x)$ para $x \leq k$,
- $s(x) = f(\langle x, s(x-1), \ldots, s(1), s(0)\rangle)$ para $x > k$.

Funciones Recursivas Parciales, VI

Recursión primitiva

A partir de las funciones totales f y g, s se define por recursión primitiva si existe k tal que:

- $s(x) = g(x)$ para $x \leq k$,
- $s(x) = f(\langle x, s(x-1), \ldots, s(1), s(0)\rangle)$ para $x > k$.

Las funciones recursivas parciales permiten definiciones por recursión primitiva.

Funciones Recursivas Parciales, VI

Recursión primitiva

A partir de las funciones totales f y g, s se define por recursión primitiva si existe k tal que:

- $s(x) = g(x)$ para $x \leq k$,
- $s(x) = f(\langle x, s(x-1), \ldots, s(1), s(0)\rangle)$ para $x > k$.

Las funciones recursivas parciales permiten definiciones por recursión primitiva.

Bosquejo de argumento:

- $\phi(x) = \pi^L(\mu y[R_{f,g,k}(\langle x.y\rangle) = 1])$, donde
- $R_{f,g,k}(\langle x.y\rangle)$, con $y = \langle y_x, \ldots, y_0\rangle$, es:

$$\forall t \leq x(t \leq k \Rightarrow y_t = g(t) \wedge t > k \Rightarrow y_t = f(\langle t, y_{t-1}, \ldots, y_0\rangle))$$

Funciones Recursivas Parciales, VII

Gödelización

Cómo lograr que las funciones recursivas parciales "sean" números:

- Empezamos por las funciones básicas:
  - $\langle 0.0 \rangle = 1$ para la identidad, $\phi_1(x) = x$ para todo x.
  - $\langle 0.1 \rangle = 2$ para la constante 1, $\phi_2(x) = 1$ para todo x.
  - ...
- El cero nos informa de que se trata de una función básica, y la segunda componente nos indica cual.
- Usamos tuplas que empiezan por 1 para indicar funciones que se obtienen por composición, por 2 para funciones que se obtienen por formación de pares, y por 3 para las que se obtienen por minimización.
  - Si $i = \langle 1.\langle j.k \rangle \rangle$, indica que $\phi_i = \phi_j \circ \phi_k$.
  - Si $i = \langle 2.\langle j.k \rangle \rangle$, indica que $\phi_i = \langle \phi_j.\phi_k \rangle$.
  - Si $i = \langle 3.j \rangle$, que ϕ_i se obtiene por minimización sobre ϕ_j.

Todos los demás números (incluido el cero) corresponden a la función totalmente indefinida: $\phi_0(x)\uparrow$ para todo x.

Funciones Recursivas Parciales, VIII

El predicado T de Kleene

(Escribimos $T(i,x,y,t)$ en vez de $T(\langle i.x.y.t\rangle)$.)

El predicado $T(i,x,y,t)$ se define como sigue: $T(i,x,y,t) = 1\ldots$

- si ϕ_i es básica, digamos, $i = \langle 0.j\rangle$, cuando $t \geq x$ y $\phi_i(x) = y$;
- si $\phi_i = \phi_j \circ \phi_k$, cuando

$$\exists s \leq t(s = \langle z.y.p.q\rangle \wedge T(k,x,z,p) \wedge T(j,z,y,q));$$

- si $\phi_i = \langle \phi_j.\phi_k\rangle$, cuando

$$y = \langle z.w\rangle \wedge \exists s \leq t(s = \langle p.q\rangle \wedge T(j,x,z,p) \wedge T(k,x,w,q));$$

- si $\phi_i = \mu\phi_j$, cuando

$$\exists s \leq t(\langle\langle x.y\rangle.s\rangle \leq t \wedge s = \langle\langle z_0.s_0\rangle,\ldots,\langle z_y.s_y\rangle\rangle \wedge$$

$$\forall m \leq y(T(j,\langle x.m\rangle,z_m,s_m)) \wedge \forall m < y(z_m \neq 1) \wedge z_y = 1).$$

Funciones Recursivas Parciales, IX

La función universal

Definimos por minimización y composición en términos de π^L y T

$$\phi_{\mathbf{u}}(i,x) = \pi^L(\mu\langle y.t\rangle[T(i,x,y,t) = 1])$$

Se cumple la siguiente igualdad: para todo i y x,

Funciones Recursivas Parciales, IX

La función universal

Definimos por minimización y composición en términos de π^L y T

$$\phi_{\mathbf{u}}(i,x) = \pi^L(\mu\langle y.t\rangle[T(i,x,y,t) = 1])$$

Se cumple la siguiente igualdad: para todo i y x,

$$\boxed{\phi_{\mathbf{u}}(i,x) = \phi_i(x).}$$

Indecidibilidad, I

El "problema de parada"

Se conoce como problema de parada dos conjuntos similares que resultan tener las mismas propiedades.

- El dominio de la función universal:

$$\mathrm{Dom}\ \phi_{\mathbf{u}} = \{\langle i.x\rangle \mid \phi_u(\langle i.x\rangle)\downarrow\} = \{\langle i.x\rangle \mid \phi_i(x)\downarrow\}.$$

- Su versión simplificada $K = \{i \mid \phi_i(i)\downarrow\}$.

Claramente: $i \in K \iff \langle i.i\rangle \in \mathrm{Dom}\ \phi_{\mathbf{u}}$: $\chi_K = \chi_{K_0} \circ \langle \phi_1.\phi_1\rangle$, luego, si $\mathrm{Dom}\ \phi_{\mathbf{u}}$ es decidible, entonces K también lo es.

Indecidibilidad, I

El "problema de parada"

Se conoce como problema de parada dos conjuntos similares que resultan tener las mismas propiedades.

- El dominio de la función universal:

$$\mathrm{Dom}\ \phi_{\mathbf{u}} = \{\langle i.x\rangle \mid \phi_u(\langle i.x\rangle)\downarrow\} = \{\langle i.x\rangle \mid \phi_i(x)\downarrow\}.$$

- Su versión simplificada $K = \{i \mid \phi_i(i)\downarrow\}$.

Claramente: $i \in K \iff \langle i.i\rangle \in \mathrm{Dom}\ \phi_{\mathbf{u}}$: $\chi_K = \chi_{K_0} \circ \langle \phi_1.\phi_1\rangle$, luego, si $\mathrm{Dom}\ \phi_{\mathbf{u}}$ es decidible, entonces K también lo es.

Y llegamos al Teorema de Indecidibilidad del Problema de Parada:

El problema de parada es indecidible.

Es decir, χ_K, la función característica de K, no es recursiva.

Indecidibilidad, II

Argumentación de la indecidibilidad del "problema de parada"

Supongamos que χ_K es recursiva total. Construimos, usándola,

$$\phi(x) = \mu z[\pi^L(\langle \chi_K(x).z \rangle) = 0]$$

que es, por tanto recursiva parcial: le corresponde algún índice j.

Esta función cumple, para todo x,

$$\phi_j(x) = 0 \text{ si } x \notin K, \text{ y}$$
$$\phi_j(x)\uparrow \text{ si } x \in K.$$

Podemos considerar el caso $x = j$: $\phi_j(j)\downarrow$ si y sólo si $j \notin K$.

Pero la definición de K nos dice: $j \in K$ si y sólo si $\phi_j(j)\downarrow$ (es decir, justamente, lo contrario).

Sólo se puede evitar la contradicción si es falsa la suposición de que χ_K es recursiva total.

Reducibilidad, I

Ya vista entre las variantes del problema de parada

Hay varias nociones de reducibilidad. Aquí consideramos sólo m-reducibilidad.

L es reducible a L', denotado $L \leq_m L'$

si existe una función recursiva total f tal que, para todo x,

- $x \in L \iff f(x) \in L'$, o sea,
- $L = f^{-1}(L')$, o sea,
- $\chi_L = \chi_{L'} \circ f$.

Reducibilidad, I

Ya vista entre las variantes del problema de parada

Hay varias nociones de reducibilidad. Aquí consideramos sólo m-reducibilidad.

L es reducible a L', denotado $L \leq_m L'$

si existe una función recursiva total f tal que, para todo x,

- $x \in L \iff f(x) \in L'$, o sea,
- $L = f^{-1}(L')$, o sea,
- $\chi_L = \chi_{L'} \circ f$.

Si $L \leq_m L'$ y L' es decidible, también lo es L.

Reducibilidad, I

Ya vista entre las variantes del problema de parada

Hay varias nociones de reducibilidad. Aquí consideramos sólo m-reducibilidad.

L es reducible a L', denotado $L \leq_m L'$

si existe una función recursiva total f tal que, para todo x,

- $x \in L \iff f(x) \in L'$, o sea,
- $L = f^{-1}(L')$, o sea,
- $\chi_L = \chi_{L'} \circ f$.

Si $L \leq_m L'$ y L' es decidible, también lo es L.

Si $L \leq_m L'$ y L es indecidible, también lo es L'.

Parametrización, I

¿Cómo obtendremos las funciones que permiten argumentar reducibilidad?

Función de parametrización o s_1

Existe una función s_1 recursiva total tal que para todo i, x, y, $\phi_i(\langle x.y \rangle) = \phi_{s_1(\langle i.x \rangle)}(y)$.

Parametrización, I

¿Cómo obtendremos las funciones que permiten argumentar reducibilidad?

Función de parametrización o s_1

Existe una función s_1 recursiva total tal que para todo i, x, y, $\phi_i(\langle x.y\rangle) = \phi_{s_1(\langle i.x\rangle)}(y)$.

Ejemplo de uso:

Sea $Tot = \{i \mid \forall x \phi_i(x)\downarrow\}$: el conjunto de los índices de funciones recursivas totales.

- Sea $\rho(\langle x.y\rangle) = \phi_{\mathbf{u}}(\langle x.x\rangle)$.
- Es recursiva parcial; por tanto, hay j tal que $\phi_j = \rho$. Fijamos ese j, y definimos $f(x) = s_1(\langle j.x\rangle)$.
- Si $x \in K$, entonces $f(x) = s_1(\langle j.x\rangle) \in Tot$ ya que, para todo y, $\phi_{s_1(\langle j.x\rangle)} = \phi_j(\langle x.y\rangle) = \phi_{\mathbf{u}}(\langle x.x\rangle) = \phi_x(x)\downarrow$
- Si $x \notin K$, entonces $f(x) = s_1(\langle j.x\rangle) \notin Tot$ ya que, para todo y, $\phi_{f(x)} = \phi_{s_1(\langle j.x\rangle)} = \phi_j(\langle x.y\rangle) = \phi_{\mathbf{u}}(\langle x.x\rangle) = \phi_x(x)\uparrow$

Parametrización, II

Interpretación como evaluación parcial

Parametrización significa evaluación parcial:

1. Tenemos un programa i del que nos importa ver su entrada como consistente en dos partes, $\phi_i(\langle x.y \rangle)$.
2. Ahora es como si tuviéramos, por ahora, sólo una de las dos partes, x.
3. Pues bien, "podemos incorporarla al programa i": $\phi_{s_1(\langle i.x \rangle)}$.
4. Y, más tarde, cuando llegue la y, podremos calcular $\phi_{s_1(\langle i.x \rangle)}(y) = \phi_i(\langle x.y \rangle)$.

La función `partial` del módulo `functools` de Python cumple el mismo rol.

Parametrización, III

Argumentación

La función s_1 recursiva total tal que para todo i, x, y, $\phi_i(x, y) = \phi_{s_1(i,x)}(y)$ es:

1. Sea $c(\langle p.q \rangle)$ tal que $\phi_{c(\langle p.q \rangle)} = \phi_p \circ \phi_q$.
2. Sean $\phi_p(x) = \langle 0.y \rangle$ y $\phi_q(\langle x.y \rangle) = \langle x + 1.y \rangle$.
3. Mediante recursión primitiva, definimos h como: $h(0) = p$ y $h(n + 1) = c(\langle q.h(n) \rangle)$.
4. Entonces, $s_1(\langle i.x \rangle) = c(\langle i.h(x) \rangle)$.

Conjuntos índex, I

Conjuntos de números o conjuntos de funciones

Ahora, un conjunto A de números naturales se puede interpretar también como un conjunto de funciones recursivas parciales, $\{\phi_i \mid i \in A\}$, es decir, como una propiedad de los programas que las calculan.

¿Cuándo es una propiedad de la función?

Si tenemos $\phi_i = \phi_j$ pero sólo uno de i y j está en A, entonces **no** es una propiedad de la función.

Conjuntos índex, II

Propiedades de programas versus propiedades de funciones

A es un conjunto índex si, para todo i y j tales que $\phi_i = \phi_j$, o bien ambos están en A o bien ninguno.

Un conjunto índex representa una propiedad de las funciones calculables que es independiente de qué programas usemos para calcularlas.

- Los índices de funciones totales, Tot,
- los índices de la función completamente indefinida, $V \ldots$

Casos triviales:

- Ninguna función cumple la propiedad ($A = \emptyset$).
- Todas las funciones la cumplen.

Casos no triviales:

Hay algún $a \in A$ y hay algún $b \notin A$.

Conjuntos índex, III

Teorema de Rice

Todo conjunto índex no trivial es indecidible.

Conjuntos índex, III

Teorema de Rice

Todo conjunto índex no trivial es indecidible.

Incluye el caso de Tot, ya visto, el de V, y muchos más.

Conjuntos índex, III

Teorema de Rice

Todo conjunto índex no trivial es indecidible.

Incluye el caso de Tot, ya visto, el de V, y muchos más.

Permutamos o no A con su complementario de manera que $0 \notin A$.

Recordemos el convenio de que ϕ_0 es la función completamente indefinida. Fijamos $a \in A$ y definimos:

$$\phi_{s_1(\langle j.x \rangle)}(y) = \phi_j(\langle x.y \rangle) = \phi_a(y) * \mathrm{sg}(1 + \phi_{\mathbf{u}}(\langle x.x \rangle)).$$

- Si $x \in K$, entonces $s_1(\langle j.x \rangle) \in A$: misma función que ϕ_a.
- Si $x \notin K$, entonces $s_1(\langle j.x \rangle) \notin A$: misma función que ϕ_0.

Y, por tanto, $K \leq_m A$.

("El `jutge` no se puede hacer de otra manera.")

Indecidibilidad, III

Volvemos al "Entscheidungsproblem"

Gödel:

- Fórmulas que definen conjuntos de números; en particular, los dominios de funciones recursivas parciales son definibles.
- En particular, hay una fórmula que define K:
  $n \in K \iff \kappa(\mathtt{n})$ es cierta.
- Números que codifican fórmulas lógicas de la aritmética.
- W: los números de las fórmulas ciertas.
- Hay una función recursiva total v que, dado n, construye el número de la fórmula $\kappa(\mathtt{n})$.

Indecidibilidad, III

Volvemos al "Entscheidungsproblem"

Gödel:

- Fórmulas que definen conjuntos de números; en particular, los dominios de funciones recursivas parciales son definibles.
- En particular, hay una fórmula que define K:
  $n \in K \iff \kappa(\mathtt{n})$ es cierta.
- Números que codifican fórmulas lógicas de la aritmética.
- W: los números de las fórmulas ciertas.
- Hay una función recursiva total v que, dado n, construye el número de la fórmula $\kappa(\mathtt{n})$.
- $n \in K \iff v(n) \in W$.

Indecidibilidad, III

Volvemos al "Entscheidungsproblem"

Gödel:

- Fórmulas que definen conjuntos de números; en particular, los dominios de funciones recursivas parciales son definibles.
- En particular, hay una fórmula que define K:
  $n \in K \iff \kappa(\mathtt{n})$ es cierta.
- Números que codifican fórmulas lógicas de la aritmética.
- W: los números de las fórmulas ciertas.
- Hay una función recursiva total v que, dado n, construye el número de la fórmula $\kappa(\mathtt{n})$.
- $n \in K \iff v(n) \in W$.

Turing:

W es indecidible.

Autoreferencia, I

Programas que se conocen a sí mismos

Existen varios enunciados que se pueden englobar en la categoría "teoremas de recursión".

Sólo mencionamos una versión:

Teorema de recursión:

Sea f una función recursiva total. Existe un número natural n tal que $\phi_{f(n)} = \phi_n$.

Nos permite tener programas que "conocen su propio texto". Por ejemplo:

- Sea $\phi_j(\langle x.y \rangle) = x$.
- Consideramos $f(x) = s_1(\langle j.x \rangle)$.
- Deducimos que hay un n tal que, para todo y,

$$\phi_n(y) = \phi_{f(n)}(y) = \phi_{s_1(\langle j.n \rangle}(y) = \phi_j(\langle n.y \rangle) = n.$$

(Aprovechamos para ver un desafío similar en Python.)

Autoreferencia, II

Relacionado con el combinador Y del lambda-cálculo

1. Mediante parametrización, obtenemos:

$$\phi_{s(x)}(y) = \phi_{\phi_x(x)}(y).$$

2. La composición $f \circ s$ es recursiva total, y tiene un índice:

$$\phi_m = f \circ s.$$

3. Y ahora, simplemente, $n = s(m)$.

Cuantificación no acotada

Por ahora, sólo existencial

Permite construir conjuntos indecidibles:

$$i \in K \iff \exists \langle y.t \rangle \, T(i, i, y, t)$$

Varias maneras equivalentes de definir los conjuntos enumerables recursivamente:

1. Cuantificación existencial ("proyección") de conjuntos decidibles.
2. Dominios de las funciones recursivas parciales.
3. Imágenes de las funciones recursivas parciales.
4. Imágenes de las funciones recursivas totales (o bien $\emptyset$).

Todo conjunto decidible es enumerable recursivamente.

K es enumerable recursivamente.

El conjunto de los números de Gödel de fórmulas demostrables es enumerable recursivamente.

Completitud

En los enumerables recursivamente

Recordemos: $L \leq_m L'$ cuando $x \in L \iff f(x) \in L'$ con f recursiva total.

- Si $L \leq_m L'$ y L' es decidible, también lo es L.
- Si $L \leq_m L'$ y L es indecidible, también lo es L'.

Completitud

En los enumerables recursivamente

Recordemos: $L \leq_m L'$ cuando $x \in L \iff f(x) \in L'$ con f recursiva total.

- Si $L \leq_m L'$ y L' es decidible, también lo es L.
- Si $L \leq_m L'$ y L es indecidible, también lo es L'.
- Si $L \leq_m L'$ y L' es enumerable recursivamente, también lo es L.
- Si $L \leq_m L'$ y L no es enumerable recursivamente, tampoco lo es L'.

Completitud

En los enumerables recursivamente

Recordemos: $L \leq_m L'$ cuando $x \in L \iff f(x) \in L'$ con f recursiva total.

- Si $L \leq_m L'$ y L' es decidible, también lo es L.
- Si $L \leq_m L'$ y L es indecidible, también lo es L'.
- Si $L \leq_m L'$ y L' es enumerable recursivamente, también lo es L.
- Si $L \leq_m L'$ y L no es enumerable recursivamente, tampoco lo es L'.

Para todo L enumerable recursivamente, $L \leq_m K$.

Complementación

El complementario de K no es enumerable recursivamente

L es decidible si y sólo si tanto L como su complementario $\overline{L}$ son enumerables recursivamente.

Complementación

El complementario de K no es enumerable recursivamente

L es decidible si y sólo si tanto L como su complementario $\overline{L}$ son enumerables recursivamente.

Por tanto, $\overline{K}$ no es enumerable recursivamente.

Grados de Indecidibilidad

"To infinity. . . and beyond!"

1. Grado cero: conjuntos decidibles.
2. Grado uno: K es enumerable recursivamente pero no decidible.
3. Grado dos: el conjunto de los índices de funciones totales $\{i \mid \forall x\, \phi_i(x)\downarrow\}$ es "el doble de indecidible": tiene con K la misma relación que $\overline{K}$ con los decidibles.
4. . . .

Grados de Indecidibilidad

"To infinity. . . and beyond!"

1. Grado cero: conjuntos decidibles.
2. Grado uno: K es enumerable recursivamente pero no decidible.
3. Grado dos: el conjunto de los índices de funciones totales $\{i \mid \forall x\, \phi_i(x)\downarrow\}$ es "el doble de indecidible": tiene con K la misma relación que $\overline{K}$ con los decidibles.
4. . . .
5. El grado de indecidibilidad de W es el límite de todos estos grados.

Contenido

Recursos Limitados

Dentro de lo decidible

- Noción de "recurso":
  1. Tiempo de cálculo,
  2. espacio de memoria requerido,
  3. otros;
  4. es posible un enfoque "abstracto" que define axiomáticamente qué es un recurso de computación.
- ¿Definición precisa de tiempo y espacio?
  - Pasos y casillas en una máquina de Turing multicinta.
  - Se pueden usar muchos otros modelos con resultados invariantes salvo factores constantes.
  - Si expresamos nuestro análisis en términos de $O(\cdot)$, el modelo de cálculo es irrelevante.

Clases de Complejidad Auxiliares

Sintaxis: $(N|D)(TIME|SPACE)(\text{bound})$

Ejemplos:

- $NTIME(n^2)$
- $DSPACE(n)$
- $DTIME(n \log n)$
- ...

Siempre en función de n: longitud de la entrada.

Clases de Complejidad Auxiliares

Sintaxis: $(N|D)(TIME|SPACE)(\text{bound})$

Ejemplos:

- $NTIME(n^2)$
- $DSPACE(n)$
- $DTIME(n \log n)$
- ...

Siempre en función de n: longitud de la entrada.

Cosas que se saben:

- (Kuroda) $NSPACE(n)$ equivale a las gramáticas sensibles al contexto.
- (Immerman / Szelepcsényi) $NSPACE(n)$, $NSPACE(\log n)$ son cerrados por complementación.
- (Savitch) $NSPACE(n) \subseteq DSPACE(n^2)$.

Clases de Complejidad Principales

- $P = \bigcup_k DTIME(n^k)$, tiempo polinómico.
- $NP = \bigcup_k NTIME(n^k)$, tiempo polinómico indeterminista.
- $PSPACE = \bigcup_k DSPACE(n^k) = \bigcup_k NSPACE(n^k)$, espacio polinómico.
- (Hay unas cuantas más.)