Resumen

Horst H. von Brand
vonbrand@inf.utfsm.cl

Departamento de Informática
Universidad Técnica Federico Santa María

Contenido

Lenguajes

Definimos *lenguajes*, planteamos nuestro interés principal: describir lenguajes (particularmente lenguajes infinitos) mediante alguna descripción finita. Obviamente interesa hallar descripciones formales simples, inteligibles; pero una descripción simple tendrá límites.

Operaciones entre palabras y lenguajes

Definimos la operación de *concatenación* entre palabras. Extendimos esa operación a lenguajes, y agregamos las operaciones de *unión* (alternancia) y *estrella de Kleene*. Usando las anteriores construimos *expresiones regulares* y los lenguajes regulares que denotan.

Autómatas finitos. No-determinismo.

Introdujimos la idea de *autómata*, un computador muy rudimentario. Describimos autómatas finitos *deterministas* (DFA) y *no-deterministas* (NFA).

No-determinismo

El no-determinismo resulta ser un concepto central: Da la posibilidad de especificar *qué* es lo que busco, sin dar el detalle de *cómo* hallarlo. En autómatas finitos simplifica muchas tareas (vea cómo de una expresión regular es muy fácil construir un NFA), más adelante es central en el problema de hallar algoritmos eficientes para grandes clases de problemas naturales (la incógnita $\mathrm{P} = \mathrm{NP}$).

Relación entre los lenguajes

Demostramos que los lenguajes aceptados por NFA y DFA y los descritos por expresiones regulares son los mismos.

Propiedades de los lenguajes regulares

Demostramos propiedades de clausura, que permiten describir en forma más simple algunos lenguajes, o que sirven para demostrar que un lenguaje no es regular (arguyendo por contradicción).

Demostramos el lema de bombeo, con el que hallamos límites a lo que un autómata finito puede lograr.

Minimizar autómatas finitos

El teorema de Mihill-Nerode da una condición necesaria y suficiente para que un lenguaje sea regular. De la demostración se desprende un algoritmo para obtener el DFA con el mínimo número de estados para un lenguaje dado.

Aplicaciones de lenguajes regulares

Notaciones inspiradas por expresiones regulares (generalmente llamadas *regex*) son comunes para expresar patrones en herramientas como editores, planillas de cálculo, programas de búsqueda en texto. Algunos lenguajes de programación incluyen *regex* integradas, para muchos otros hay bibliotecas (incluso estandarizadas) que ofrecen esta funcionalidad.

Agrupar caracteres en símbolos como operadores, identificadores y palabras reservadas, reconocer comentarios en un lenguaje de programación (la fase de *análisis léxico*) es cómodo de expresar mediante *regex*.

Gramáticas

Para extender las capacidades de lenguajes regulares, definimos *gramáticas*. Las clasificamos (junto a los lenguajes generados) en la *jerarquía de Chomsky*.

De particular interés (por su simplicidad) son gramáticas y lenguajes *de contexto libre*.

Gramáticas y lenguajes de contexto libre

Definimos *árboles de derivación* como una forma compacta de resumir conjuntos de derivaciones.

Aplicaciones asocian significado a la estructura del árbol de derivación, con lo que gramáticas que dan más de un árbol de derivación para alguna palabra (*gramáticas ambiguas*) son indeseables.

Autómatas de pila

Autómatas finitos se ven limitados por su memoria finita (sus estados). Les adosamos una pila como memoria, dando PDAs. Hay dos formas naturales de definir si el PDA M acepta: *por estado final*, dando el lenguaje $\mathscr{L}(M)$; *por pila vacía*, dando el lenguaje $\mathscr{N}(M)$.

Lenguajes aceptados por estado final o pila vacía por un PDA y lenguajes generados por gramáticas de contexto libre son los mismos.

Hay lenguajes aceptados por PDA no-deterministas que ningún PDA determinista acepta.

Propiedades de los lenguajes de contexto libre

Vimos formas de simplificar gramáticas de contexto libre. Definimos la *forma normal de Chomsky* (CNF), que simplifica demostraciones y algoritmos.

Demostramos propiedades de clausura, que permiten describir en forma más simple algunos lenguajes, o que sirven para demostrar que un lenguaje no es de contexto libre (arguyendo por contradicción).

Demostramos el lema de bombeo, con el que hallamos límites a lo que un autómata de pila puede lograr.

Aplicaciones de gramáticas de contexto libre

Las gramáticas de lenguajes naturales son «casi» de contexto libre.

La introducción de gramáticas de contexto libre para describir (parcialmente) la sintaxis de lenguajes de programación significó una revolución de proporciones.

Hay programas (*compiladores de compiladores*) que toman una gramática (y acciones asociadas) y construyen un programa que reconoce el lenguaje.

XML es una manera popular de describir formatos flexibles. Es basado en gramáticas de contexto libre.

Máquinas de Turing

Construimos *máquinas de Turing* como descripción simple de «computación». Se inventaron varios formalismos adicionales, que rápidamente se demostraron equivalentes.

La tesis de Church dice que computación es equivalente a lo que hace una máquina de Turing.

Máquinas de Turing

Vimos varios modelos extendidos: múltiples cintas, múltiples pistas, ...

Esbozamos la construcción de una máquina de Turing universal, que dada la codificación de una máquina de Turing y sus datos de entrada simula el funcionamiento de la máquina sobre los datos.

Máquinas de Turing deterministas y no-deterministas aceptan los mismos lenguajes.

Máquinas de Turing

Definimos el lenguaje L_d, demostramos que ninguna máquina de Turing lo acepta.

Problemas y reducciones

Un *problema* es un lenguaje. La pregunta del caso es: ¿es $\sigma \in L$?

Una *reducción* del problema $A \subseteq \Sigma^*$ al problema $B \subseteq \Gamma^*$ es una función $f : \Sigma^* \to \Gamma^*$ tal que $f(\sigma) \in B$ si y solo si $\sigma \in A$. Se anota $A \leq B$.

Computabilidad

Definimos *lenguajes computables* (o *decidibles*), aceptados por máquinas de Turing que siempre se detienen, y *lenguajes computablemente enumerables*, aceptados por máquinas de Turing (que puede que nunca se detengan). Exhibimos ejemplos de lenguajes computablemente enumerables que no son decidibles. Exhibimos lenguajes que no son computablemente enumerables.

Discutimos la clasificación de un lenguaje y su complemento. Solo algunas combinaciones son posibles.

Computabilidad y reducciones

- Si $A \leq B$, y A no es decidible, B no es decidible.
- Si $A \leq B$, y A no es computacionalmente enumerable, B no es computacionalmente enumerable.

Computabilidad

Varios problemas pendientes (¿estas gramáticas generan en mismo lenguaje?, ¿es ambigua esta gramática?) resultan ser no decidibles.

Computabilidad

El teorema de Rice dice que ninguna propiedad no trivial del lenguaje aceptado por una máquina de Turing es decidible.

Algoritmos eficientes

Definimos *algoritmos eficientes* como algoritmos deterministas que resuelven un problema en tiempo acotado por un polinomio en el tamaño de los datos de entrada.

Las clases P, NP y coNP

El problema A está en P si hay un polinomio p tal que A es aceptado por una máquina de Turing determinista que da a lo más $p(|\sigma|)$ pasos al procesar σ.

El problema B está en NP si hay un polinomio p tal que B es aceptado por una máquina de Turing (no necesariamente determinista) que da a lo más $p(|\sigma|)$ pasos al aceptar σ. Los pasos de la máquina no-determinista se entienden como el camino más corto que lleva a aceptar.

El problema C está en coNP si $\overline{C}$ está en NP.

Las clases P, NP y coNP

Es claro que $P \subseteq NP$, $P \subseteq NP \cap coNP$.

Hay mucha evidencia que indica que $P \neq NP$, bastante de que $NP \neq coNP$ y que $P \neq NP \cap coNP$.

Problemas y reducciones polinomiales

Una *reducción polinomial* del problema $A \subseteq \Sigma^*$ al problema $B \subseteq \Gamma^*$ es una función $f : \Sigma^* \to \Gamma^*$ tal que $f(\sigma) \in B$ si y solo si $\sigma \in A$, y $f(\sigma)$ puede ser computada por una máquina de Turing que da a lo más $p(|\sigma|)$ pasos, donde p es un polinomio. Se anota $A \leq_p B$.

Los «tiempos» que toman distintos modelos de cómputo están relacionados por polinomios. Podemos obviar el modelo exacto.

Problemas NP-completos

Un problema H es NP-duro si todos los problemas en NP se reducen polinomialmente a él.

Un problema C es NP-completo si está en NP y es NP-duro.

Para demostrar que A es NP-completo:

Demostrar que está en NP: Para $\sigma \in A$ hay un certificado $c = c(\sigma)$ tal que un algoritmo determinista verifica en tiempo acotado por un polinomio en $|\sigma|$ que $\sigma \in A$.

Demostrar que es NP-duro: Hallar un problema NP-completo C tal que $C \leq_p A$.

Problemas NP-completos

- Si el problema pide dividir en dos conjuntos, considere SAT o Partition.
- Si pide asignar rótulos de un conjunto pequeño, o dividir en un número constante de subconjuntos, considere coloreo de grafos.
- Problemas de asignación de recursos suelen ser variantes de Knapsack, Job Shop o Bin Packaging.
- Si busca ordenar objetos en cierto orden, considere Hamiltonian o TSP.

Problemas NP-completos

- Si interesa un conjunto mínimo, intente Vertex Cover.
- Si interesa un conjunto máximo, pruebe con Clique o Independent Set.
- Al dividir objetos en muchos conjuntos pequeños, vea 3D-Match.
- Si el número 3 aparece naturalmente en el problema, considere 3SAT, 3Color o 3D-Match (no, no es broma).
- Si todo falla, recurra a 3SAT o directamente a una máquina de Turing.

Decisión y búsqueda/optimización

Si el problema A es NP-completo, el problema de búsqueda u optimización respectivo puede resolverse con un número polinomial de soluciones de A.

Resumen

- Dimos una rápida mirada a algunas áreas enormes.
- En ramos sucesivos verán técnicas para lidiar con problemas complejos.