Projecto II

Análise e Síntese de Algoritmos

2011/2012

Relatório de Projecto

62462 Miguel Fonseca
<miguelcsfonseca@ist.utl.pt>

I. Introdução

Este relatório descreve uma solução para o problema do *parser* de sequências binárias descrito no enunciado, centrada no algoritmo de Cocke–Younger–Kasami (CYK). É feita uma breve análise da complexidade da solução, referindo nuances trazidas pela implementação concreta da solução, escrita em C e usando estruturas de dados simples.

II. Descrição do problema e sua solução

O enunciado especifica que a definição de um gerador de sequências binárias se faz combinando regras de apenas duas formas: $A \rightarrow BC$ e $A \rightarrow \alpha$, onde A, B e C são geradores e α é ou 0 ou 1. Em terminologia de teoria de linguagem formal, um gerador é uma gramática, A, B e C são símbolos não-terminais, e α é um símbolo terminal. Acontece que uma gramática composta inteiramente de regras nestas formas diz-se estar na Forma Normal de Chomsky (FNC). O algoritmo CYK, operando exclusivamente sobre gramáticas dadas nesta forma, adequa-se plenamente à solução do problema posto. De facto, determinar se uma sequência binária é produzida por um gerador corresponde a fazer o reconhecimento da sequência com um *parser* compilado a partir das regras gramaticais especificadas.

CYK é um bom exemplo de programação dinâmica. Tira partido da FNC na medida em que vê cada regra como ou um caso terminal ($A \rightarrow \alpha$) ou um caso recursivo ($A \rightarrow BC$). No caso terminal, reconhecer uma regra é tão somente reconhecer um símbolo terminal – no caso em questão, 0 ou 1. No caso recursivo, um gerador A é composto por dois geradores mais simples B

e C; assim, reconhecer A é reconhecer B seguido de C. O algoritmo guarda assim em memória, para cada sub-gerador, informação sobre se uma subsequência do *input* pode ser gerado por ele.

Isto leva a outra característica do algoritmo, que é a de ele ter em conta todas as subsequências do *input* possíveis para reconhecer regras. Para isso, começa a ler da primeira posição do *input* e faz o *parsing* das subsequências de comprimento *1*, deslocando-se para a direita. Passa então para as subsequências de comprimento *2* e por aí fora. Para aquelas de comprimento superior a *1*, considera-se ainda as suas partições possíveis, *e.g.* para a subsequência $a_1a_2a_3a_4$, estuda-se $a_1+a_2a_3a_4$, $a_1a_2+a_3a_4$ e $a_1a_2a_3+a_4$.

No cerne do algoritmo está uma estrutura de dados capaz de lidar com a memorização de todas estas combinações: o reconhecimento de todas as possíveis regras da gramática para todas as possíveis subsequências do *input*. Trata-se de uma matriz $Int \times Int \times Int \rightarrow Bool$, a que nos referiremos por $P[i,j,k]$. A matriz contabiliza, com *verdadeiro* ou *falso*, se uma dada subsequência do *input* pode ser gerada por uma dada regra; i corresponde ao índice do início da subsequência, j ao seu comprimento e k ao índice da regra.

Com esta estrutura, é claro como o algoritmo emprega programação dinâmica para funcionar eficientemente. Após inicializar P com todos os elementos a *falso*, percorre, num primeiro tempo, o *input* à procura de regras R_k unitárias ($A \rightarrow \alpha$), marcando $P[i,1,k]$ como *verdadeiro* quando aplicável – *e.g.* se $R_k \rightarrow 0$, então, para qualquer i tal que $input[i]$ == '*0*', $P[i,1,k] \leftarrow$ *verdadeiro*. Passada essa fase, resta encontrar regras compostas ($A \rightarrow BC$), estudando todas as subsequências como descrito anteriormente. Para tal, usa-se P em cada partição possível da seguinte forma: para uma regra $R_A \rightarrow R_BR_C$, vê-se se a primeira parte da partição é gerada por B e a segunda por C, *i.e.*, se $P[j,k,B]$ e $P[j+k,i-k,C]$ são verdadeiros. Nesse caso, A gera a subsequência coberta pela composição das subsequências da partição, e define-se $P[j,i,A] \leftarrow$ *verdadeiro.*

Como o algoritmo vai das subsequências mais curtas às maiores, garante que, ao chegar à maior subsequência – o *input* todo –, encontra uma partição (A,B) para a regra inicial $R \rightarrow AB$ caso o *input* seja gerado por R. No final, saber se o *input* é gerado por R é saber o valor de $P[1,n,k]$, onde n é o comprimento do *input* e *k* o índice da regra inicial.

O programa implementado inclui ainda alguma lógica para além do algoritmo, em particular para fazer a leitura do texto de entrada, coleccionar e contar as regras definidas, para fazer as alocações necessárias à construção do *parser*. Finalmente, o *parser* é chamado para cada linha de *input* e imprime-se “yes” ou “no” consoante o resultado do caso.

III. Análise da solução e sua implementação

O programa age em dois tempos: uma fase de preparação e uma de execuções sucessivas do algoritmo.

Na fase de preparação, guardam-se, à medida que se lê de *standard input*, as regras unitárias e compostas em duas pilhas enquanto se conta U e C, o número de regras respectivas. De seguida, conhecendo-se esses tamanhos, aloca-se espaço para um vector de regras unitárias e um de compostas, para os quais se transfere o conteúdo das pilhas. Representando o número total de regras $U+C$ por R, esta fase tem assim complexidade, tanto espacial como temporal:

$\Theta(R)$.

A seguir, uma execução individual do algoritmo CYK faz-se em duas fases: uma para o reconhecimento de regras unitárias, a outra para as compostas.

A primeira percorre cada carácter do *input* e procura correspondê-lo às regras unitárias. Assim sendo, para um *input* de tamanho N, tem-se uma complexidade temporal de:

$\Theta(U \times N)$.

A segunda fase olha para cada possível partição do *input* (partições em que nenhuma das partes é de tamanho nulo) e procura correspondê-la às regras compostas. Representando o número de partições possíveis para um *input* de tamanho N por *part(N)*, tem-se a complexidade:

$\Theta(C \times part(N))$.

As partições fazem-se iterando sobre o comprimento possível de uma subsequência de tamanho superior a 1, *i.e.*, $N-1$ vezes. Para cada comprimento, itera-se sobre as posições de início de partição possíveis: para subsequências de tamanho 2, itera-se $N-1$ vezes; para tamanho 3, $N-2$ vezes, etc. Para cada posição de início, itera-se sobre as partições possíveis a partir dela. Sem procurar obter uma fórmula explícita para $part(N)$, é fácil ver que os números de iterações para estes três ciclos são majorados por N. Assim, podemos dizer que:

$$part(N) = O(N^3),$$

donde a complexidade do algoritmo:

$$\Theta(U \times N) + \Theta(C \times part(N)) = O(N \times (U + C \times N^2)).$$

No pior caso, tem-se $R = C$ e uma complexidade $O(R \times N^3)$. Contas feitas, a execução do programa inteiro, considerando um conjunto de *inputs* I_i de tamanhos respectivos N_i, tem como desempenho, no pior caso:

$$\Sigma_i [O(R \times N_i^3)] = O(R \times \Sigma_i [N_i^3]).$$

Em termos de complexidade espacial, há três estruturas a considerar: os vector de regras unitárias, de tamanho $\Theta(U)$, o de regras compostas, de tamanho $\Theta(C)$, e a matriz $P[i,j,k]$, de tamanho $\Theta(N \times N \times K)$. Para conter o estritamente necessário, ter-se-ia $K = R$; no entanto, escolheu-se fixar K ao número máximo de regras que o formato de entrada suporta: 26, de acordo com as letras do alfabeto. Isto permite simplificar a execução ao evitar fazer mapeamentos mais custosos entre regras e índices na matriz: associa-se uma letra maiúscula L directamente ao seu índice, que vai de 0 a 25; em ASCII, basta para tal subtrair 65 ao valor do carácter. Em resumo, tem-se a complexidade espacial:

$\Theta(R + N^3)$, que em teoria seria de $\Theta(R + N^3 \times R)$.

IV. Resultados da avaliação experimental

O programa funciona como esperado, passando os testes fornecidos para o efeito e recebendo nota inteira no sistema de submissão Mooshak. Da bateria, o teste `t08`, com *14* regras, um *input* de *224* caracteres e um de *125* caracteres, é executado em *2,12 s* de processamento numa máquina de 32 bits com um *core* de *1,6 GHz*. O exame da execução pela ferramenta Valgrind revela que são feitas *66 891* alocações de memória, num total de *1 997 698 bytes*. Por comparação, o teste `t02`, com *7* regras e *3 inputs* de tamanhos *19*, *20* e *11*, é executado em tempo de processamento inferior a *0,01 s*, e em tempo real de *0,005 s*, fazendo *1 064* alocações num total de *30 921 bytes*.

V. Referências

en.wikipedia.org: artigos “Chomsky normal form”, “CYK algorithm”