Anaísa Forti da Fonseca
11811ECP012

Trabalho 02 - Autômatos de Pilha

Neste segundo laboratório, foi solicitada a implementação de um Simulador Universal de Autômatos de Pilha Não-Determinísticos (APs). O programa aceita a especificação de um AP e, dentre uma lista de cadeias de caracteres dada, diz quais pertencem e quais não pertencem à linguagem reconhecida pelo autômato.

As entradas consistem nas informações necessárias para montar o autômato, tais como o número de estados, os símbolos terminais (alfabeto), os símbolos de pilha, os estados de aceitação e as transições. Juntamente às especificações do autômato, são fornecidas as cadeias de entrada. A saída trata de dizer se tais cadeias pertencem ou não à linguagem do autômato, imprimindo “aceita” ou “rejeita” para cada cadeia que é analisada.

Por sua simplicidade e rapidez em desenvolvimento, a solução foi implementada em python. O código interpreta as entradas fornecidas pelo usuário com as informações para montar o autômato e armazena estas informações em variáveis e em estruturas de dados como a lista. Cada transição do autômato é uma lista com 5 elementos: o primeiro é o estado de origem, o segundo é o símbolo terminal, o terceiro é o símbolo de pilha que deve estar no topo da pilha, o quarto é o estado de destino e o quinto e último elemento é o símbolo de pilha a ser colocado na pilha. Todas as transições são armazenadas em uma única lista. O mesmo raciocínio vale para as cadeias: cada uma é uma lista e todas são armazenadas em uma outra lista (lista de listas).

O programa possui a função “percorreCadeia” que recebe como parâmetros uma cadeia dentre as fornecidas pelo usuário, o estado inicial do autômato (sempre 0) e o símbolo inicial da pilha (Z). Esta é uma função que, para cada caractere da cadeia sendo percorrida, verifica todas transições possíveis por meio de um laço “for” e, através da recursividade, consegue percorrer toda a lista de transições para cada símbolo da cadeia (e cadeias vazias também). Por poder ter mais de uma transição possível a qualquer momento (incluído transições vazias), a cadeia pode percorrer sequências de transições diferentes e, portanto, ter múltiplos estados finais possíveis quando é completamente percorrida.

A cada transição realizada, um ou mais símbolos de pilha são empilhados (ou desempilhados). Para que esta parte do código não ficasse se repetindo em todas as diferentes condições, foi criada uma função chamada “empilhamento”, que trata

exclusivamente disso. Ela analisa se a transição empilha ou desempilha os símbolos de pilha e altera a pilha de acordo. Também, é nela que a recursividade da função “percorreCadeia” é chamada para continuar percorrendo a cadeia.

Quando o fim da cadeia é atingido ou quando a cadeia é vazia, a função “percorreCadeia” compara o estado final com os estados de aceitação do autômato. Caso algum estado final seja de aceitação, a função retorna “true”. Em contrapartida, caso não haja nenhum estado final que seja de aceitação, a função analisa se ainda há alguma transição vazia possível para realizar e chegar a novos estados finais. Se estes continuarem não sendo de aceitação, o programa retornará “false”.

Para analisar o retorno da função “percorreCadeia”, foi utilizado um laço de repetição “for” verificando se, para cada cadeia, o retorno foi “true” ou “false”. Para os retornos “true” o laço imprimirá “aceita”, mostrando que a cadeia pertence à linguagem reconhecida pelo autômato. Analogamente, o laço imprimirá “rejeita” para os retornos “false”, revelando que a cadeia não pertence à linguagem.

Com relação ao funcionamento e qualidade, a solução apresentou-se eficiente retornando o resultado esperado para 100% dos casos de teste, sendo estes alguns retirados da internet e os do próprio sistema de correção.

Para que fosse possível analisar a eficiência do programa em termos de tempo, foi utilizada a notação Big-O. Considerando que a função principal possui um laço “for” que percorre a lista de t transições e é feita baseada em recursividade, a cada símbolo lido da cadeia esse laço “for” é feito novamente. Assim, considerando o pior caso possível, baseando-se na notação Big-O e desconsiderando toda e qualquer transição vazia, pode-se definir o tempo como $O(t^n)$, sendo t o número de transições do autômato e n o número de símbolos da cadeia. A notação é feita desta maneira já que toda a lista de transições precisa sempre ser percorrida para cada símbolo da cadeia. Isso se aplica a AP’s não-determinísticos sem transições vazias.

Conclui-se, então, que quanto maior a cadeia, mais tempo será necessário para percorrer ela completamente. Transições vazias não foram consideradas neste caso pelo fato de que o pior caso possível está sendo considerado e, então, o programa poderia entrar em loop, realizando apenas transições vazias e nunca terminar de percorrer a cadeia, seja lá qual fosse seu tamanho. Analogamente, também é possível definir um tempo para AP’s determinísticos: como para cada símbolo da cadeia há apenas uma transição possível, o tempo necessário é igual à quantidade de símbolos na cadeia, ou seja, O(n). E, da mesma maneira que em AP’s não-determinísticos, quanto maior a cadeia maior será o tempo para percorrê-la.

Já para análise da eficiência do programa em termos de espaço, observou-se o tamanho máximo da pilha atingido. Na pilha podem ocorrer três ações: empilhamento, desempilhamento ou nenhum dos dois. Como o pior cenário possível está sendo levado em conta, considerou-se que houve apenas empilhamentos nas transições e partiu-se de um AP determinístico. Nele pode haver, então, apenas empilhamento de um símbolo de pilha a cada transição. Como o autômato é determinístico, há apenas uma transição possível para cada símbolo da cadeia, ou seja, há apenas um empilhamento a cada símbolo da cadeia. Portanto, lembrando que a pilha nunca está vazia (sempre está pelo menos com o símbolo inicial Z), o tamanho máximo da pilha atingido em um AP determinístico é n+1, sendo n o número de símbolos na cadeia e o +1 referente ao símbolo inicial da pilha.

Seguindo este raciocínio para um AP não-determinístico sem transições vazias e novamente considerando o pior cenário possível, todas as transições seriam possíveis para cada símbolo e cada um destes empilharia um símbolo de pilha. Nesta análise tem-se novamente em mente que, para ver a eficiência do programa em termos de espaço, são levados em conta apenas empilhamentos na pilha, e não desempilhamentos.

Quando uma cadeia de n símbolos é percorrida, o número máximo de empilhamentos possíveis é n e todos os estados finais possíveis (considerando caminhos diferentes para chegar em cada um) podem ser dados por t^n, sendo t o número de transições do autômato. Então, quando o autômato percorrer toda a cadeia, para cada caminho diferente percorrido, tem-se no final uma pilha individual (relativa a cada estado final atingido por caminhos diferentes) de tamanho n.

Como a quantidade de estados finais possíveis neste raciocínio é dado por t^n, pode-se concluir que uma pilha contendo todas as pilhas individuais de tamanho n tratadas acima teria um tamanho de nt^n. Tendo em mente que na solução implementada não foi feita uma pilha das pilhas individuais, considera-se a seguir apenas a individual de tamanho n. Pensando em termos de espaço e tamanho de cadeia, conclui-se que quanto maior a cadeia, maior será a pilha individual de cada estado final. Transições vazias novamente não foram consideradas pelo mesmo motivo: no pior caso possível o programa poderia entrar em loop realizando apenas transições vazias e ele não só nunca terminaria de percorrer a cadeia como também faria empilhamentos a cada transição (vazia ou não), fazendo com que o tamanho final de cada pilha individual tendesse a infinito.

Com isso, examinando os dados de espaço e tempo analisados, encontra-se um ponto muito importante em comum: quanto mais caracteres entram no AP, maior o tempo e o espaço necessários para percorrer essa cadeia e para armazenar a pilha, respectivamente.