Projeto 1 – Análise e Síntese de Algoritmos

LEIC-A - Instituto Superior Técnico 2017/2018

Grupo al058
Rui Alves 65284
João Coelho 86448

Introdução

Este relatório faz parte dos entregáveis e suporta o código produzido para a implementação do primeiro projeto da cadeira de ASA, do curso de LEIC no Instituto Superior Técnico (Alameda), ano letivo 2017/2018.

O problema apresentado pelo corpo docente tem todo um contexto que vai ser ignorado ao longo deste documento, sendo que nos vamos focar apenas no seu significado prático: tendo um grafo dirigido, pretende-se determinar quantas Componentes Fortemente Ligadas o grafo possui (doravante designadas por *SCCs*, do inglês *Strongly Connected Components*), quantas ligações (doravante designadas por *pontes*) existem entre as SCCs (ignorando múltiplas ligações entre o mesmo par de SCCs), e, finalmente, apresentar essas pontes, que devem ter a forma "idOr idDst", onde *idOr* é o identificador da SCC de origem e *idDst* é o identificador da SCC de destino. O identificador de uma SCC é o mínimo da lista de vértices que a constituem, sendo que os vértices do grafo são identificados pelo subconjunto dos números naturais de 1 a V, onde V é o número de vértices do grafo. Por último, convém referir que a lista de pontes deve ser apresentada por ordem crescente, primeiro pelo *idOr* e depois pelo *idDst*.

O *input* que o programa deve receber é o número de vértices do grafo (V), o número de arestas do grafo (E) e a lista exaustiva dessas mesmas arestas com a forma "vOr vDst", onde *vOr* é o identificador do vértice de origem e *vDst* o identificador do vértice de destino. Esta lista de arestas não tem de estar ordenada.

Descrição da Solução

A linguagem de programação utilizada foi C. Esta decisão foi tomada tendo em conta não só a maior proficiência dos elementos do grupo nesta linguagem em comparação com as outras alternativas (C++ e Java), como também pelo facto de nos permitir gerir mais facilmente a alocação de memória durante a sua execução, sendo que esse recurso está limitado no ambiente de testes do projeto (*mooshak*).

Ao receber o *input*, é gerada a lista de adjacências do grafo, implementada como uma lista de listas ligadas. Sobre essa lista de adjacências é aplicada a nossa implementação do algoritmo de *Tarjan*.

O algoritmo de *Tarjan* recebe um grafo dirigido e retorna as suas SCCs. A sua ideia fundamental consiste em fazer uma pesquisa em profundidade primeiro (*DFS*, do inglês *Depth First Search*) no grafo, visitando apenas cada vértice uma vez e guardando o menor valor de profundidade pertencente à sua descendência. Durante a sua execução, é guardado se cada vértice já foi visitado, sendo também adicionado a uma pilha quando isso acontece. É com a gestão dessa pilha, ao longo do algoritmo, que se determinam as SCCs do grafo.

A nossa implementação tem algumas modificações, pois não nos interessa guardar uma estrutura de dados com os vértices pertences a cada SCC, mas sim a que SCC pertence cada vértice (numeradas de 0 a N-1, onde N é o número de SCCs) e para cada SCC, o seu mínimo (identificador). O primeiro valor é guardado no primeiro elemento de cada lista ligada (cada vértice), o segundo é guardado numa lista própria para esse fim (variável *minSCC* no código). Assim, conseguimos saber com complexidade O(1) o identificador da SCC a que cada vértice pertence. Com essa informação conseguimos então criar uma lista de possíveis pontes, percorrendo a lista de adjacência. Aqui fala-se em *possíveis* pontes pois há a possibilidade de haver pontes que aparecem mais que uma vez, para os casos em que há mais de uma aresta que liga o mesmo par de SCCs.

De seguida, procede-se à ordenação dessa lista de possíveis pontes, primeiro em função do vértice de destino e depois em função do vértice de origem, usando o algoritmo de ordenação *Counting Sort*. Uma das razões que nos levou a utilizar este algoritmo é o facto de ser estável, ficando assim a lista com a ordem pretendida, já mencionada anteriormente, na introdução deste documento.

Por último, a lista de possíveis pontes é iterada duas vezes: uma primeira para contar o número de pontes, ignorando as repetições, e uma segunda para as imprimir. De referir que é fácil ignorar as repetições tendo a lista ordenada, basta comparar com o valor anterior (se forem iguais, é repetido). Convém também mencionar que, ao iterarmos a lista de adjacências para criar a lista de possíveis pontes, a memória alocada para a lista de adjacências vai sendo libertada, minimizando assim a memória total alocada num dado momento, nunca chegando a ter as duas estruturas, na sua totalidade, simultaneamente em memória.

Análise Teórica

Teoricamente e analisando o código, o algoritmo tem uma complexidade linear, tanto em termos de complexidade espacial como temporal.

- Complexidade temporal: O(V+E)

Ao longo de todo o código, os ciclos são sempre O(V) ou O(E), à exceção do já mencionado passo em que a lista de arestas é criada a partir da lista de adjacências, tendo assim complexidade O(V+E) ao percorrer a lista toda. O algoritmo de *Tarjan* tem também complexidade O(V+E), se o grafo de *input* estiver representado como uma lista de adjacências – daí termos escolhido esta estrutura como representação, ao invés de uma matriz de adjacências – e desde que testar se um determinado vértice está na pilha possa ser feito em tempo constante, como acontece no nosso caso ao guardarmos essa informação para cada vértice nos nós iniciais de cada lista ligada.

Existia também a possibilidade de implementarmos o algoritmo de *Kosaraju* para descobrir os SCCs, que também tem complexidade O(V+E). No entanto, optamos por implementar o algoritmo de *Tarjan* visto que este é um pouco mais rápido: ambos têm o mesmo comportamento assimptótico, no entanto, no algoritmo de *Kosaraju* existem algumas constantes multiplicativas que são ignoradas na análise assimptótica (por serem constantes) mas que podem fazer com que o tempo de execução do algoritmo seja maior.

Por último, o algoritmo *Counting Sort* tem também complexidade linear porque não é à base de comparações, outra das razões pela qual foi escolhido.

- Complexidade espacial: O(V+E)

A estrutura de dados mais complexa que utilizamos é a lista de adjacências, que tem complexidade O(V+E). As restantes estruturas como a lista de possíveis pontes ou a lista de identificadores de cada SCC tem complexidade O(E) ou O(V), respetivamente. A lista de identificadores terá exatamente tamanho V caso todos os vértices sejam uma SCC de um único vértice (caso particular em que todos os vértices estão ligados sequencialmente) e a lista de possíveis pontes terá exatamente tamanho E também nesse caso, sendo que todas as arestas são de facto pontes.

Análise Experimental

Para a análise experimental do programa foram-nos disponibilizados alguns testes.

O gráfico 1 foi desenhado com o máximo de memória alocada em função de V+E. De mencionar que esse máximo foi obtido através da ferramenta *massif* do *Valgrind*, visto que libertamos alguma memória a meio da execução do programa, ao invés de libertar a memória toda no fim.

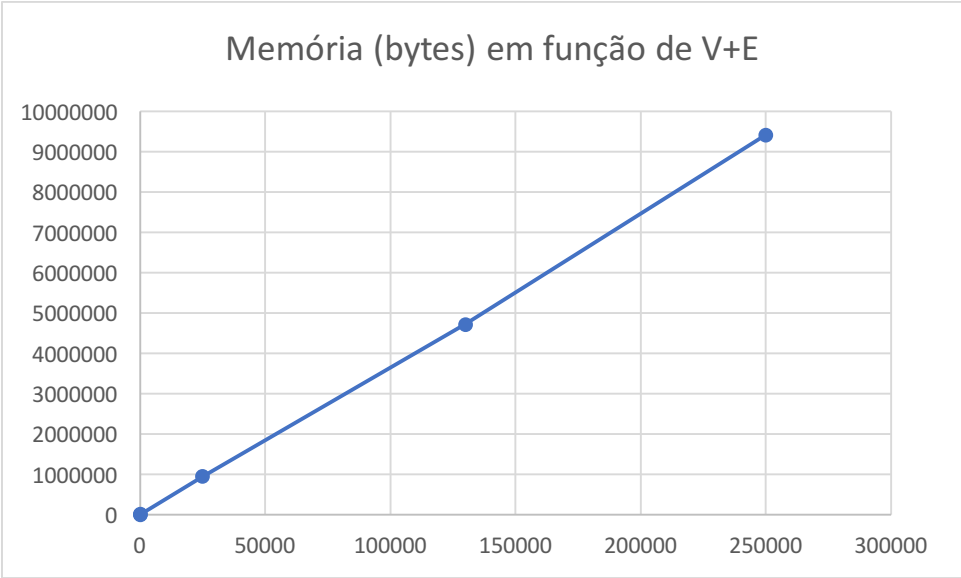

Gráfico 1 – Memória alocada (máximo) em função de V+E

O gráfico 2 foi desenhado com base na média de tempo decorrido para correr 10 vezes cada teste, também em função de V+E. Estes tempos foram obtidos com a função *time* do *Linux*.

Gráfico 2 – Tempo de execução em função de V+E

Analisando os gráficos, conclui-se que a memória é linear, indo de acordo ao esperado através da análise teórica. No entanto, relativamente ao tempo, este parece ter um comportamento quadrático, que pode ter várias explicações: poderá ter-nos escapado alguma operação que não seja constante nos ciclos referidos, falta de precisão da função *time* ou ainda a influência de outros processos que estejam a correr paralelamente na máquina.

Referências

Bibliográficas:

- Introduction to Algorithms, Third Edition: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein September 2009 ISBN-10: 0-262-53305-7; ISBN-13: 978-0-262-53305-8

Para além da obra anterior, foram consultados os seguintes websites:

- https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm
- https://www.geeksforgeeks.org/counting-sort/
- https://en.wikipedia.org/wiki/Counting_sort