TÉCNICO LISBOA

Análise e Síntese de Algoritmos

1º Projeto 2013/2014 – Grupo 34

70577 João Godinho
73951 Pedro Silva

Introdução

No âmbito da cadeira de Análise e Síntese de Algoritmos foi-nos proposto que resolvêssemos um problema relacionado com partilha de informação entre pessoas. Cada pessoa tem um conjunto de outras pessoas com quem partilha o que recebe, formando assim grupos de partilha. A tarefa que nos foi atribuída envolvia a classificação dos grupos formados pelas partilhas. Especificamente o número de grupos máximos de pessoas que partilham informação (1), o tamanho do maior desses grupos (2) e o número de grupos que apenas partilham informação dentro do próprio grupo (3).

Solução

Depois de analisarmos o *input* e *output* dos ficheiros de exemplo, decidimos equiparar as pessoas a nós e as partilhas a arcos, formando assim um grafo. Esta equiparação levou-nos a pensar numa solução que implementa grupos fortemente ligados (SCC). Deste modo podemos extrapolar o *output* com SCCs da seguinte forma:

1. Número de SCCs;
2. SCC com mais elementos;
3. Número de SCCs que não partilham informação para fora do SCC.

Para podermos calcular os pontos referidos, recorremos ao algoritmo de Tarjan, a decisão do uso do Tarjan sobre o algoritmo de Kosajaraju deveu-se ao Tarjan apresentar uma complexidade menor em termos práticos.

A implementação direta do Tarjan permitiu-nos responder de imediato ao ponto 1, para respondermos aos pontos 2 e 3 foi necessário adicionar funcionalidades ao algoritmo.

A nossa solução foi desenvolvida em Java, de modo que tivemos que implementar certos objetos que nos permitiram aplicar o algoritmo de Tarjan. Para que mais facilmente se entendam os objetos implementados, segue-se uma breve descrição de como o algoritmo de Tarjan funciona; a ideia principal é percorrer os nós do grafo e os seus vizinhos, cada vez que se visita um nó, atribui-se um índice (incrementado após cada atribuição) e um *lowlink* (inicializado com o valor do índice) e coloca-se o nó na pilha de nós visitados. Após as atribuições visitam-se os vizinhos do nó atual e verifica-se se cada um dos vizinhos já foi visitado ou se se encontra na pilha: se não foi visitado, entra-se nesse nó e repete-se o processo anunciado; se se encontra na pilha atribui-se ao *lowlink* do nó atual o mínimo entre o seu *lowlink* e o índice de cada vizinho; se já foi visitado e não está na pilha, é porque pertence a outro SCC e é ignorado. Depois de se visitar o nó vizinho, atribui-se o mínimo entre o *lowlink* do nó atual e o índice do nó visitado. Depois de percorrer todos os vizinhos, verifica-se se o *lowlink* do nó atual é igual ao seu índice, caso seja verdade é porque estamos num grupo fortemente ligado, portanto fazemos *pop* da pilha até tirarmos o nó atual onde estamos.

Os objetos que implementámos no nosso projeto para satisfazer o algoritmo foram *Node* e *Stack*, estes encontram-se implementados da seguinte maneira:

- Node
  - int index → índice do nó

    - int lowLink → *lowlink* do nó
    - int scc → SCC a que este nó pertence
    - ArrayList<Integer> neighbors → lista de vizinhos
- SimpleStack
    - int top → posição do *top* da pilha
    - int topFake → posição temporária do *top* da pilha;
    - int stack[]→ pilha onde cada posição tem o nó de tamanho N-1 nós.
    - boolean isInStack[]→ vetor onde cada posição indica se o nó está na pilha, entre 0 e N-1 nós.

A nossa implementação começa por fazer o tratamento do ficheiro de entrada, criando um vetor de N-1 nós do tipo *Node* onde o nó *x* fica colocado na posição *x-1*, e a cada *Node* é adicionado uma lista de vizinhos depois de tratar as arestas. É também inicializada uma instância de *SimpleStack*, com um tamanho de N-1 nós nos atributos *stack* e *isInStack,* este atributo permite que a procura na pilha tenha uma complexidade de *O(1)*, uma vez que apenas é necessário aceder ao vetor *isInStack* para saber se um nó se encontra na pilha. Com esta descrição é-nos possível responder ao número de SCCs contando as vezes que num nó o *lowlink* é igual ao índice.

Para responder à segunda linha do *output* apenas foi necessário adicionar um contador que incrementa sempre que se faz um *pop* da pilha e no final do ciclo guardar o maior entre o contador atual e o maior grupo encontrado anteriormente (variável *biggestGroupSize* inicializada a 0).

Por fim, de modo a responder à última alínea tivemos que criar um segundo ciclo que simula um *pop* da pilha, ao nó que é retirado atribui-se o SCC a que este pertence (variável *scc* do objeto *Node*), no fim do primeiro ciclo temos atribuído aos nós o SCC a que estes pertencem. Posto isto, realiza-se o segundo ciclo, igual ao anterior, mas que serve para verificar se cada nó que se retira tem os vizinhos todos dentro do seu SCC, se todos os nós a que se fez *pop* verificam a condição referida, então consideramos um SCC fechado, incrementando a variável *nClosedGroups*.

Análise Teórica

O projeto encontra-se dividido em 2 fases: o tratamento do ficheiro de entrada e o cálculo dos grupos fortemente ligados. Na fase do tratamento do ficheiro de entrada, o projeto terá uma complexidade $O(2V) = O(V)$, onde V representa o número de nós. Na fase de cálculo dos grupos fortemente ligados, o projeto terá uma complexidade de $O(V + E)$, onde V representa o número de nós e E o número de partilhas (arcos). Esta complexidade seria mais alta se não tivéssemos otimizado a procura de um nó no *stack* com o vetor adicional *isInStack*, como detalhado em cima. Se a procura fosse feita analisando todos os elementos do *stack* e verificando se algum deles corresponde ao nó pretendido, a complexidade cresceria para $O(V^2 + E)$.

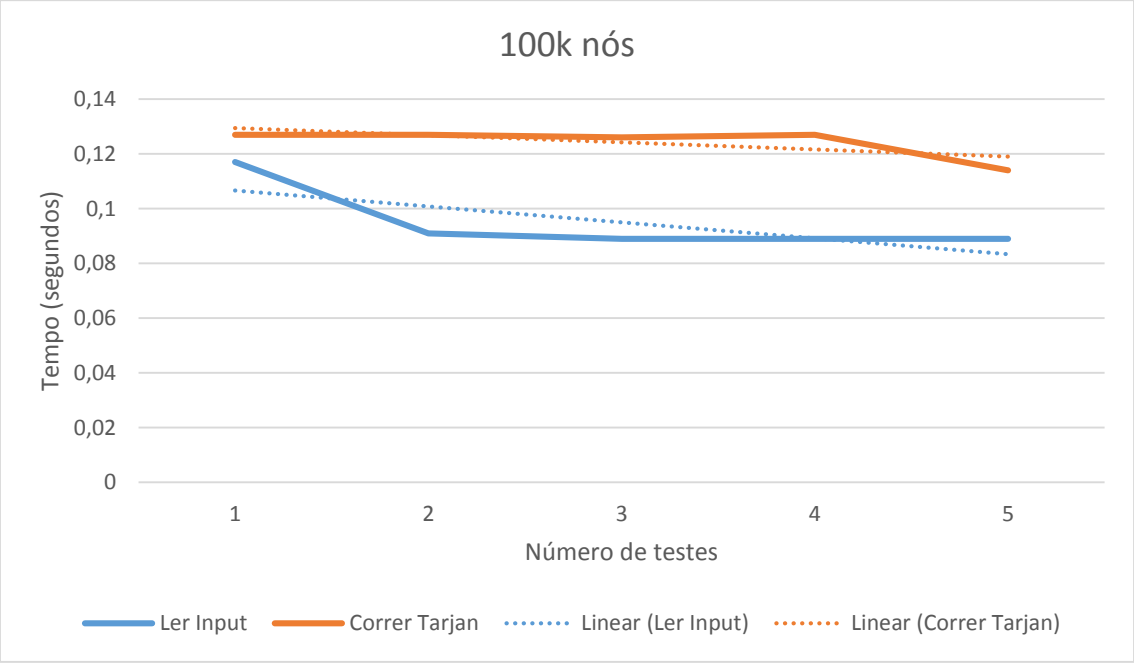
O projeto no total terá portanto uma complexidade $O(V + V + E) = O(2V + E) = O(V + E)$.

Avaliação Experimental

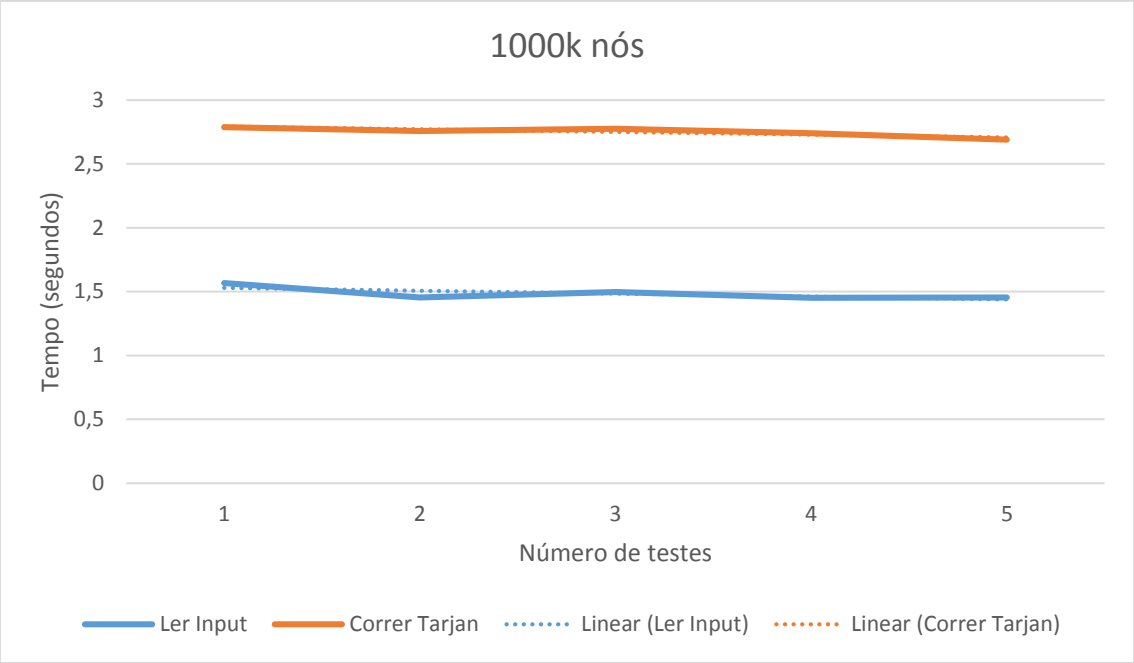
Para testarmos o tempo que a nossa implementação demorava a correr *inputs* de diferentes tamanhos, criámos uma pequena aplicação que recebendo um número, criava esse mesmo número de arcos, ou seja, dado o número 4, a aplicação devolvia: 5 4 | 1 2 | 2 3 | 3 4 | 4 5.

Foram criados 3 ficheiros com 100k, 1000k e 10000k nós, cada teste foi corrido 5 vezes numa máquina com um *i5@2.40GHz* com *4GB* de memória num sistema operativo *Kubuntu 13.10* com o *JDK7u51* e a *JVM* foi configurada para um *heap size* de *1GB* e memória máxima de *2GB*, além disto, os testes foram corridos em linha de comandos sem tarefas que fossem dispensáveis (*e.g.:* ambiente gráfico).

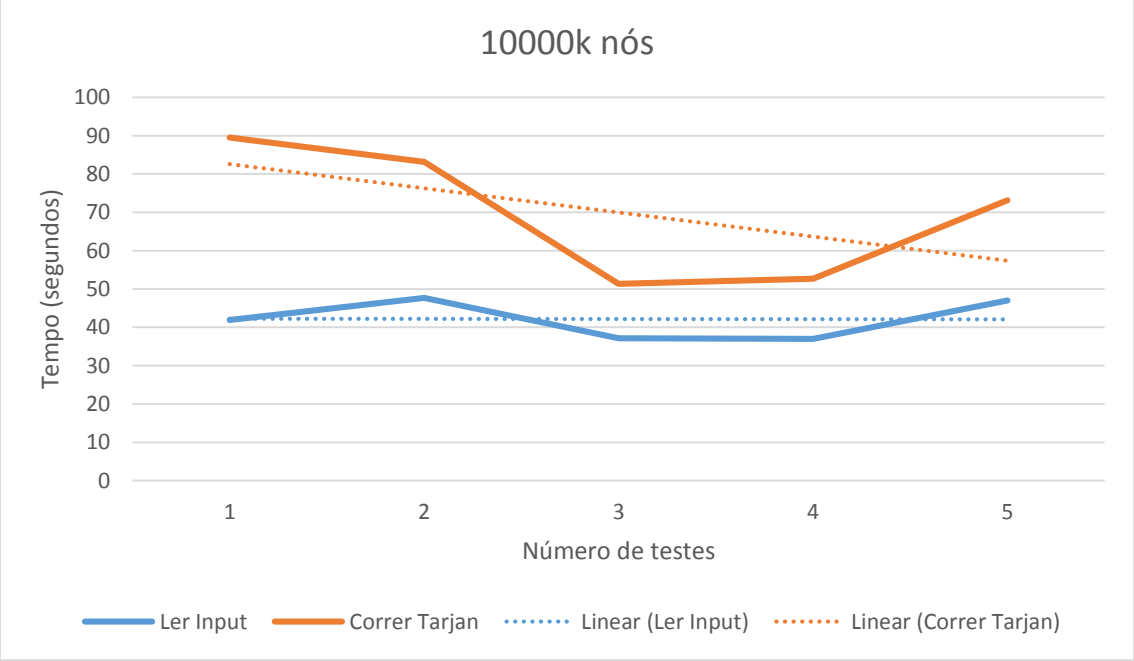
Foram medidos os tempos de tratamento do *input* assim como os tempos de correr o algoritmo de Tarjan modificado.



Ficheiro 1



Ficheiro 2



Ficheiro 3