Relatório - Lab 01

Sabrina Beck Angelini - RA 157240

Descrição

O algoritmo monta os grafos com listas de adjacências, verifica se o grafo da blueprint danificada possui todos os vértices da blueprint do arquivo morto, marcando uma flag como *true* em cada vértice em comum (na blueprint danificada). Depois disso, para cada vértice da blueprint danificada que está presente no arquivo morto, faz-se uma busca em profundidade procurando por outros vértices presentes no arquivo morto que possuem um caminho formado por apenas novos vértices. Para cada vértice do arquivo morto encontrado, a aresta equivalente a esse caminho do arquivo morto é excluída. Se ambas forem da mesma cidade, o grafo do arquivo morto ficará sem nenhuma aresta, ou seja, para cada aresta do arquivo morto entre v_1 e v_2 existe um caminho entre v_1 e v_2 na blueprint danificada formado apenas por vértices novos.

Análise

Seja n_1 o número de vértices da blueprint antiga, m_1 o número de arestas da blueprint antiga, n_2 o número de vértices da blueprint danificada e m_2 o número de arestas da blueprint danificada.

O programa principal chama a função **buildGraph**() que pode ser conferida na linha *177*. Vamos analisar a complexidade dela primeiro. Essa função lê os grafos de entrada, sendo uma leitura o número m de arestas de um grafo e m iterações para ler cada aresta, sendo que cada leitura resulta em na leitura de dois vértices. Como estamos construindo o grafo, os vértices lidos na iteração corrente podem já estar no grafo graças à leitura anterior, logo, iteramos por todos os vértices já presentes no grafo até encontrá-los, no pior caso, já teremos lido todos os vértices e os que estamos lendo atualmente estão no final da lista, logo essa iteração terá complexidade $O(n)$, sendo n o número de vértices no grafo. Com isso, conseguimos analisar que a complexidade de **buildGraph**() é $O(mn)$. Também temos algumas chamadas à classe list da biblioteca padrão do C++, **list::begin**(), **list::end**() e **list::push_back**() que possuem complexidade $O(1)$. O programa principal chama essa função duas vezes, uma para a blueprint antiga (linha *31*) e um para a blueprint danificada (linha *33*), isso nos dá uma complexidade de $O(m_1n_1 + m_2n_2)$.

Vamos analisar o método **remove**() na linha *50*. Esse método utiliza as chamadas **list::begin**() e **list::end**() com complexidade $O(1)$ e **list::erase**() com complexidade linear no número de elementos removidos, que no caso é um só, logo $O(1)$. Nós iteramos todos os vértices da blueprint do arquivo morto, e para dois deles, percorremos as listas de adjacências (que possuem no máximo $n_1 - 1$ arestas), logo a complexidade será $3n_1 = O(n_1)$.

Agora vamos analisar **depthSearchVisit**(vf, v, G) na linha *79*. Dado um vértice fonte da blueprint danificada (a chamada na linha *157*), percorremos todos os seus vértices adjacentes que, no pior caso, seriam $2m_2$ iterações contando com as chamadas recursivas ($2m_2$ vezes que a condição da linha *91* vai executar). Temos que **remove**() é chamado m_1 vezes para remover, no máximo, todas as arestas do arquivo morto. Temos assim a complexidade $O(m_2 + m_1n_1)$.

A análise de **checkVertexes**(G1, G2), na linha *108*, itera nos vértices da blueprint do arquivo morto n_1 e, para cada um desses vértices, itera os n_2 vértices da blueprint danificada, resultando em $O(n_1n_2)$.

O programa principal também chama a função **areFromSameCity**(G1, G2) que está na linha *137*. Ela itera sobre todos os vértices da blueprint danificada, ou seja, n_2 iterações (linha *144*). Há uma nova iteração para cada vértice dessa mesma blueprint para marcá-lo como não lido (linha *151*) e uma chamada ao método recursivo **depthSearchVisit**(vf, v, G) com uma complexidade de $O(n_1^2 + n_1(m_2 + m_1n_1))$. Fora da iteração temos uma chamada ao método **checkVertexes**(G1, G2) nos dando mais a complexidade $O(n_1n_2)$ dessa chamada e o último *for* (linha *163*) que verifica cada vértice da blueprint do arquivo morto $O(n_1)$. Tudo isso resulta em $O(n_1^2 + n_1(m_2 + m_1n_1) + n_1n_2 + n_1)$.

Para o programa principal, temos $O(n_1^2 + n_1(m_2 + m_1n_1) + n_1n_2 + n_1 + m_1n_1 + m_2n_2)$. Sabemos que $n_1 \leq n_2$, pois a blueprint danificada deve conter todos os vértices do arquivo morto. Logo, podemos majorar $n_1 = O(n_2)$.

$$O(n_1^2 + n_1(m_2 + m_1n_1) + n_1n_2 + n_1 + m_1n_1 + m_2n_2)$$
$$O(n_2^2 + n_2(m_2 + m_1n_2) + n_2n_2 + n_2 + m_1n_2 + m_2n_2)$$
$$O(n_2^2 + n_2m_2 + m_1n_2^2 + n_2^2 + n_2 + m_1n_2 + m_2n_2)$$
$$O((1 + m_1 + 1)n_2^2 + (n_2 + 1 + n_2 + n_2)m_2)$$
$$O(m_1n_2^2 + m_2n_2)$$