Trabalho Prático 1 - Aritmofobia

Filipe de Araújo Mendes - 2021031920
as
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

flipeara@ufmg.br

1. Introdução

O trabalho prático 1 está relacionado a grafos e como encontrar a menor distância entre dois vértices em um grafo. Mais especificamente, o TP1 requer que seja encontrado o menor caminho entre o primeiro vértice e o último, com a utilização de pesos, passando, necessariamente, por um número par de arestas e não passando por nenhuma aresta de peso ímpar.

2. Modelagem

2.1. Grafo

Tendo em mente que o algoritmo teria que percorrer os vértices adjacentes a outro vértice, assim como a não garantia de que os grafos teriam n² arestas, escolhi fazer o grafo com uma lista de adjacência, já que facilitaria para encontrar os vizinhos de um vértice e não teria espaço desperdiçado, assim como a matriz teria, por não existir muitas arestas incidentes a um vértice.

Já que o grafo tem tamanho fixo, conhecido no início da execução do programa, foi decidido que os vértices seriam representados em um vector, que possui uma forma eficiente de acesso aos elementos, podendo utilizar vector[i] para acessar a posição desejada.

As arestas, por sua vez, teriam que ser lidas uma por uma e adicionadas ao grafo, portanto, vector não foi visto com a melhor opção, pela necessidade de realocação sempre que ele atinge o tamanho máximo, assim, foi escolhida a estrutura list para representar as arestas adjacentes a um vértice. Tal escolha não foi prejudicial na hora de acessar as arestas de um vértice, já que elas foram acessadas sequencialmente.

Por fim, a aresta precisava representar duas coisas: o outro vértice e a distância entre os dois vértices. Por esse motivo, foi escolhida a estrutura pair para conseguir guardar todas essas informações dentro das lists de arestas. Assim, o grafo foi montado como um vector<list<pair<int,int>>>.

2.2. Algoritmo:

Levando em conta que o algoritmo não poderia levar em consideração as arestas de peso ímpar do grafo, foi feita uma verificação na hora de montar o grafo, se o peso fosse par, a aresta seria adicionada ao grafo, caso contrário, ela era ignorada. Assim, todos os grafos teriam, necessariamente, só arestas de peso par.

Para calcular a distância entre o primeiro e o último vértice, foi utilizada uma adaptação do algoritmo de Dijkstra. Em Dijkstra, olhamos o vetor distância e pegamos a menor distância e colocamos o vértice como visitado, assim, temos o primeiro vértice como visitado e com distância 0 para chegar nele. Então, olhamos todos os vértices alcançáveis pelo vértice 1 com 1 aresta entre eles e atualizamos o vetor distância com as menores distâncias encontradas para novamente pegar o vértice com menor distância e marcá-lo como visitado.

Isso funciona, pois não há como encontrar uma distância menor para um vértice visitado depois de marcá-lo, já que todos os outros caminhos para chegar nele a partir de outro vértice tem pelo menos a mesma distância.

O algoritmo utilizado segue a mesma lógica. Existe o vetor distância, que é sempre atualizado também. Entretanto, a maneira de atualizar as distâncias é levemente diferente: ao invés de atualizar a distância dos vértices alcançados com 1 aresta de distância do vértice v, é atualizada a distância dos vértices com 2 arestas de distância do vértice v. Por exemplo, se temos o vetor distância como: 1(0), 2 (∞), 3(∞). Supondo que exista uma aresta do 1 para o 2 com distância 4 e uma aresta do 2 para o 3 com distância 2, o algoritmo calcularia a distância da aresta 1-2-3, que seria:

$$4 + 2 = 6$$

Assim, atualizaria o vetor distância para: 1(0), 2(∞), 3(6). Veja que o vértice 2 ainda não tem distância para ser alcançado. Isso acontece pois ainda não foi encontrado nenhum caminho com um número par de arestas.

Assim, o algoritmo calcula as distâncias de 2 em 2 arestas, então ao final, encontrará a menor distância entre o primeiro e o último vértice andando por um número par de arestas.

Para isso, o algoritmo pega o vértice com menor distância no vetor de distâncias, por meio da utilização de uma fila de prioridades e marca esse vértice como visitado. Então, ele varre todos os vizinhos de v e, para cada vizinho, varre todos os vizinhos dos vizinhos de v, e então atualiza as distâncias no vetor distância de todos os vizinhos dos vizinhos de v.

Para a complexidade do algoritmo, vejamos um pseudocódigo simplificado:

```
1.vetorDistância[n]
2.Adicionar primeiro vértice para fila de prioridade
3.N vezes:
4.      Pegar primeiro elemento na fila de prioridade
5.      Marcar v
6.      Para cada vizinho w de v
7.              Para cada vizinho x de w
8.                      Se x nao visitado e d[v] + c(v,w) + c(w,x) < d[x]
9.                              D[w] = d[v] + c(v,w) + c(w,x)
10.                             Fila de prioridade adiciona (x,d[x])
```

Assim, teremos que o algoritmo faz n vezes o loop da linha 3, mas na soma dessas n vezes, ele só pode executar a linha 6 um total de m vezes, que é o número de arestas do grafo, assim como na linha 7. Tendo assim uma complexidade de $O(m^2 * \log n)$.

A complexidade, portanto, é pior do que a de dijkstra, por precisar olhar mais combinações no grafo, mas consegue executar em um tempo bom, para grafos não tão cheios.