Trabalho Prático II
Algoritmos I

Vinicius Silva Gomes[1]

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

vinicius.gomes@dcc.ufmg.br

Abstract. *This article describes steps to troubleshoot a resource flow problem in a pre-provided path system. The objective is to develop a program that is able to obtain a path and maximize the flow of resources through it, since this flow is limited by the road with the least capacity on the path. It will be documented from the analysis of the problem and its inputs to the solution reached and its properties.*

Resumo. *Este artigo descreve os passos para a solução de um problema de fluxo de recursos em um sistema de caminhos previamente fornecido. O objetivo é desenvolver um programa que seja capaz de obter um caminho e maximizar o fluxo de recursos através dele, uma vez que esse fluxo é limitado pela rodovia com menor capacidade no caminho. Será documentado desde a análise do problema e suas entradas até a solução alcançada e suas propriedades.*

1. Introdução

O enunciado contextualiza um problema de logística de fluxo de suprimentos para ajudar as vítimas de desastres naturais. É solicitado o desenvolvimento de um software que, dado o sistema rodoviário do estado, retorne uma rota que maximize o peso que pode ser transportado por um caminhão entre um par de cidades informadas e que estejam cadastradas no sistema rodoviário fornecido.

Sempre haverá um caminho entre as duas cidades pelas quais o caminhão deve alcançar. No entanto, calcular qual seria a capacidade máxima de recursos que o caminhão pode transportar, visto que essa capacidade é definida pela rodovia que suporta o menor peso no caminho, e levando em consideração as diversas rotas que podem existir entre as duas cidades, pode não ser uma tarefa trivial.

Baseado nisso, este artigo tem como objetivo documentar a solução para este problema, enfatizando como foi feita a Modelagem Computacional do cenário descrito, quais as características e algoritmos utilizados na Implementação e qual a Complexidade Assintótica desses algoritmos.

2. Modelagem Computacional

Para modelar o problema, foram feitas análises a respeito das entradas que ele admite e do retorno esperado para cada conjunto de entrada. Primeiramente, é possível observar que o caráter do problema indica que precisaremos computar caminhos no sistema rodoviário fornecido e analisar uma característica específica desses caminhos, de modo a determinar

qual o mais adequado. Dito isso, uma forma prática de representar o sistema rodoviário seria através de um grafo, visto que ele representa muito bem as relações que desejam ser estabelecidas entre as cidades e possui uma grande variedade de algoritmos para computar distâncias e caminhos entre os vértices do grafo (pode ser que algum desses algoritmos auxilie na resolução do problema proposto).

Para o grafo do sistema rodoviário, temos que os vértices são as cidades do sistema e as arestas são as rodovias, sendo que essas arestas são direcionadas: pode ser que exista um caminho de u para v, mas não exista de v para u; e ponderadas, onde o peso l_e é o peso máximo dos caminhões que podem passar pela rodovia.

Com a estrutura adequada para representar o sistema rodoviário, é preciso identificar como computar os caminhos e, sendo mais específico que isso, como computar o caminho que maximize o peso dos caminhões que vão levar os recursos de uma região a outra. Para tanto, existe um algoritmo conhecido que computa caminhos em um grafo direcionado e ponderado (arestas com peso $l_e \geq 0$): o algoritmo de Dijkstra. Contudo, ele computa o menor caminho de um nó fonte s para um nó de destino t ou para todos os outros nós do grafo [Kleinberg and Tardos 2005], e não a aresta que maximize a capacidade no caminho dos dois vértices.

Apesar disso, é possível adaptar o algoritmo de Dijkstra para que ele compute o que desejamos, alterando a função de relaxamento do algoritmo. Assim, tentaremos maximizar o tamanho das arestas inseridas ao longo do caminho entre os nós s e t. Para tal, ao invés de mantermos uma variável que guarda o tamanho total do caminho até um nó u, vamos manter o tamanho da menor aresta de s até u. Assim, sempre que um outro caminho que "aumente o tamanho" dessa aresta for encontrado, ela será substituída e, ao final, será possível saber qual a banda máxima de todos os possíveis caminhos entre os nós s e t.

Portanto, após definir qual a melhor estrutura de dados para representar o sistema rodoviário e traçar uma estratégia para computar o que o problema pede, podemos discutir sobre como o algoritmo soluciona o problema proposto e sobre os aspectos específicos da implementação do programa, na seção 3.

3. Implementação

O programa foi desenvolvido na linguagem **C++** e compilado pelo compilador **G++**, da **GNU Compiler Collection**. Algumas variáveis de entrada aparecerão com frequência nos pseudocódigos: V e E, o número de cidades e rodovias (vértices e arestas) que compõem o grafo G que representa o sistema rodoviário informado; e l_e o fluxo máximo de recursos que podem passar por uma aresta. A seguir, os principais algoritmos do programa serão discutidos.

3.1. `main()`

A função `main` recebe uma entrada do problema, armazenando o grafo e as consultas que serão executadas em matrizes do tipo `vector<vector<pair<int, int>>>` e `vector<pair<int, int>>`, respectivamente. Após isso, ela percorre a matriz das consultas, pegando cada par fonte-destino e envia para a função `maximum_bandwith_dijkstra()`, que computa a aresta que maximiza o peso no caminho dos dois vértices e exibe o resultado calculado.

Para representação do grafo foi utilizado a estrutura `vector` em conjunto com a estrutura `pair`, que computam um vetor e uma tupla de valores de tipo genérico definidos na declaração. O vetor realiza o acesso a uma posição em $\mathcal{O}(1)$ e oferece alguns facilitadores, como os métodos de manipulação dos seus elementos, que já foram previamente implementados. A estrutura `pair` será adequada pois facilita a representação das arestas do grafo, tendo um elemento da tupla como o vértice de destino da aresta e o outro como peso l_e da aresta em questão.

Para representar as *queries*, a estrutura `pair` também será interessante pois torna a representação de um par fonte-destino mais semântica e prática no código.

3.2. `maximum_bandwith_dijkstra()`

Esse é o principal algoritmo do programa, sendo ele o responsável por calcular a maior banda possível no caminho entre dois nós. Se trata de um algoritmo guloso, inspirado na abordagem utilizada pelo algoritmo de Dijkstra para computar menores caminhos entre um nó fonte e os outros nós do grafo. O código vai tentar fazer as melhores escolhas locais, torcendo para que essas melhores escolhas locais reflitam na melhor solução global.

O algoritmo utiliza um vetor do tipo `vector<int>` para armazenar as bandas máximas no caminho de s até cada nó u do grafo G. A banda máxima para um vértice i está contida na i-ésima posição do vetor de bandas máximas.

O vetor de bandas é inicializado com 0 para todos os vértices do grafo, com exceção da fonte, que é inicializada com um valor infinito. Considerando o domínio do problema, o valor "infinito" é 1000000, o valor máximo que cada aresta $1 \leq l_e \leq 100000$ pode admitir. Isso foi feito pois, inicialmente, assumimos que uma quantidade infinita de recurso pode chegar até a fonte mas não conhecemos o tamanho das arestas para os outros nós que existem no grafo. Portanto, inicializamos a banda da fonte como infinito, a banda dos outros vértices como 0 e iterativamente vamos atualizando as bandas dos outros vértices do grafo, a medida que caminhos com bandas maiores são encontrados.

Além desse vetor, o algoritmo faz uso de uma fila de prioridade, que será baseada num heap binário que ordena a partir do maior valor presente na lista. Essa estrutura será inicializada com o nó fonte e será útil para que as maiores arestas (as de maior prioridade) sejam processadas primeiro. Essa é a melhor escolha local da abordagem gulosa do algoritmo, que vai priorizar o processamento das arestas maiores e, com isso, parece também contribuir para que a banda máxima seja a maior possível nesses caminhos.

A fila de prioridade será implementada por uma estrutura de tipo `priority_queue<pair<int, int>>`, na qual o primeiro elemento do par é a banda do nó i no momento da inserção na fila (irá auxiliar na ordenação da fila de prioridade, para que ela processe os nós com maior banda primeiro) e o segundo elemento é o próprio nó i.

Dessa forma, enquanto a fila de prioridade possuir algum nó para processar, essa aresta u será removida da fila e os vizinhos do nó de destino dela serão analisados. Para cada vizinho v, se a aresta que conecta u a v ou a banda máxima do caminho de s até u forem maiores que a banda máxima para o nó v naquela iteração, a banda de v será atualizada para o mínimo entre a banda de u e o tamanho da aresta que conecta u e v e a nova banda máxima de u até v junto com o nó v serão inseridos na fila de prioridade.

Em suma, o que essa iteração faz é percorrer o grafo a partir das bandas máximas e procurar caminhos que aumentem a banda de algum nó naquela iteração. Se o caminho não aumentar a banda, ele será descartado e o caminho atual (que possui banda máxima maior) será mantido.

Em dado momento, não haverá um caminho que maximize a banda de algum vértice. Nesse momento, a fila ficará vazia e o algoritmo irá retornar a banda máxima obtida para o nó de destino. O pseudocódigo 1 mostra esse algoritmo que computa o caminho com a banda máxima para cada nó.

Algorithm 1 Computa a banda máxima do caminho $s \rightarrow$ todos os nós do grafo G

Input: Conjunto de vértices V, o grafo G e o par de nós fonte s e destino t
Output: Capacidade l_e que representa a banda máxima do caminho entre s e t

```
PQ ← ∅
for cada v ∈ V de G do
    Bandwith[v] ← 0
end for
Bandwith[s] ← ∞
PQ ← (∞, s)
while PQ ≠ ∅ do
    u ← RETIRA-TOPO(PQ)
    for cada vizinho v de u do
        b ← MIN(Bandwith[u], capacidade(u, v))
        if Bandwith[v] < b then
            Bandwith[v] ← b
            PQ ← (b, v)
        end if
    end for
end while
return Bandwith[t]
```

O algoritmo não finaliza a execução após computar a banda para o nó t pois esse valor está sujeito a atualizações em qualquer iteração do algoritmo. Dessa forma, retornar o resultado de maneira apressada para "economizar" execução poderia acarretar num retorno incorreto por parte do algoritmo.

Assim, o código calcula todas as bandas do grafo até finalizar a execução do *loop* principal, e então retorna o valor correto da capacidade máxima do nó t.

4. Análise de Complexidade

Para definir a complexidade do algoritmo, serão usados alguns padrões para se referir as variáveis do problema. Iremos denotar V e E como o número de cidades e rodovias (vértices e arestas), respectivamente, do grafo G, que representa o mapa do sistema rodoviário; e denotaremos Q como o número de *queries* executadas em uma instanciação do problema.

O algoritmo é composto, em suma, por uma função que computa o fluxo máximo de recursos entre s e t a partir do caminho que maximiza essa banda. Para tal, foi usado

uma implementação que segue uma premissa semelhante a do algoritmo de Dijkstra, tanto na filosofia do algoritmo quanto na estrutura do código, se comparado às implementações tradicionais desse algoritmo de menores caminhos (essencialmente é o mesmo algoritmo, mas a função de relaxamento foi modificada para computar outro tipo de distância).

Assim, como para o algoritmo de Dijkstra, a complexidade do algoritmo proposto é limitada pelas operações implementadas pela fila de prioridade e pelas iterações ao longo do algoritmo, sendo elas envolvendo os vértices e as arestas do grafo e o número de *queries* executadas para aquela instância do problema.

O primeiro passo do algoritmo é declarar o vetor de bandas máximas dos vértices com todos os valores iguais a zero. Essa tarefa é feita pela estrutura `vector` na sua inicialização, com complexidade assintótica de $\mathcal{O}(V)$, visto que todas as V posições do vetor precisam ser acessadas e terem seus valores zerados.

Além disso, a fila de prioridade usada será baseada em um heap binário, que permite que operações de inserção e de retirada do primeiro elemento (maior banda máxima) sejam feitas em $\mathcal{O}(\log V)$ [Cormen 2012]. Assim, como todos os nós serão processados pelo menos uma vez em algum momento da execução e, no pior caso, todas as arestas melhoram o valor da banda máxima de algum nó (o que significa, em termos práticos, que todas elas serão processadas pelo algoritmo no *loop* principal e a nova banda, junto com seu respectivo vértice, serão inseridos na fila de prioridade), a complexidade da iteração principal do algoritmo é da ordem de $\mathcal{O}(E \cdot \log V)$.

Portanto, para encontrar a banda máxima para um par de vértices fonte-destino, o algoritmo tem complexidade total de $\mathcal{O}(V + E \cdot \log V)$. Como podem ser executadas Q consultas de banda máxima, a complexidade assintótica final para uma instanciação do problema é de $\mathcal{O}(Q \cdot (V + E \cdot \log V))$.

5. Compilando o Código

Para compilar o código, foi usado um `makefile` com o código:

```
all:
    g++ main.cpp -Wall -o tp02
```

Desse modo, caso o `makefile` não esteja disponível no ambiente de testes, o comando `g++ main.cpp -Wall -o tp02` deverá ser utilizado para compilar, e o comando `./tp02 < ./test_cases/1.txt` para executar o programa com alguma entrada de exemplo "1.txt".

Referências

Cormen, T. H. (2012). *Algoritmos - Teoria e Prática*. GEN LTC, 3th edition.

Kleinberg, J. and Tardos, E. (2005). *Algorithm Design*. Addison-Wesley, 1th edition.