Documentação TP2

Gabriel Bordoni (2018050715)

Algoritmos I
Universidade Federal de Minas Gerais

21 de dezembro de 2021

1 Resumo do Problema

Após o último problema da Black Friday, a empresa varejista voltou a precisar de uma solução para o seu negócio. Desta vez, já focada na parte da logistíca da distribuição de seus produtos, a empresa pediu para que fosse criado um algortítmo que otimizasse seu custo de conectar suas n lojas por meio de seus meios de transporte disponíveis, os drones, as motos e os caminhões.

Na entrada do problema, seriam passado então a quantidades de lojas que a varejista possuia, assim como a quantidade de drones disponíves e o limite máximo de distância cujo uma moticleta poderia percorrer. Com essas informações, seria então responsabilidade do algorítimo gerar, a partir de um conjunto de posições, um caminho que conectasse as lojas ao mesmo tempo que gerasse um custo mínimo, por sua vez dado pela multiplicação dos quilometros rodados de cada meio nas conexões multiplicados pelo custo de cada um deles (também fornecidos como valores iniciais).

2 Modelagem e Implementação

Considerando que o problema poderia ser modelado como um grafo adirecional conectado, pois cada vértice teria acesso a qualquer outro e vice e versa, temos que a solução procurada pela empresa passa diretamente pela determinação de uma árvore geradora mínima para o sistema. Ao fazer isso, o segundo passo que deve ser analisado é a logística dos meios de transportes. Isso pois, sendo o drone isento de custo e a motocicleta limitada superiormente por uma quilometagem, temos que devemos procurar as maiores arestas da árvore para empregar os drones ao mesmo tempo que procuramos empregar as motocicletas nas restantes que estejam em seu limite de operação. Feito isso, certemente obteremos ao final do algorítmo a solução desejada, que é o menor custo de logística da varejista.

Para a obtenção da árvore gerado mínima, foi decidido a utilização do algorítimo de Kruskal com a otimização da utilização do Union-Find. O algorítimo de Kruskal representado pela classe que leva seu nome recebe as diversas posições fornecidas e gera delas os vétices e arestas do problema por meio das estruturas denominadas Node e Edge. Cada vértice gerado é armazenado em uma lista assim como as arestas, que são armanadas em uma espécie de matriz de adjacencia achatada, mas além delas também temos uma lista de conjutos, os Set. Essa estrutura de dados nada mais é que uma lista encadeada que tem como objetivo fazer com que os vértices armazenados nelas tenham uma chave em comum, o que é responsável por garantir a aciclicidade da solução. Inicialmente temos uma lista de conjuntos pois inicialmente temos que cada vértice está em um conjunto diferente, mas a medida que vamos decobrindo os menores caminhos que os ligam, um conjunto vai se unindo ao outro.

Após lido a posição de todas as lojas e inicializado todas as estruturas de dados que abstraem o problema, a primeira coisa feita, seguindo a proposta de Kruskal, foi realizar a ordenação da lista de arestas por meio do comprimento que cada uma representava. O método utilizado para isso foi o MergeSort, e sua escolha se embasou no fato dele ser um algoritmo ótimo para ordenação ao mesmo tempo que é de simples implementação, **Algorithm 1**. O custo de memória é um ponto negativo da escolha feita, mas acreditando que esse recurso não é algo crítico para esse tipo de problema, foi-se optado por fazer assim da mesma forma.

Com a matriz de adjacência unidimencional ordenada, o segundo passo mais importante na solução do problema foi encontrar deles um conjunto de arestas que conectassem todos os vétices por meio do conceito de Union-Find, **Algorithm 2**. Nessa hora o que se foi feito foi, para cada aresta em linha crescente de distancia, era verificado se os vértices em seu extremo participavam do mesmo grupo ou não (pela chave que o Set dá a cada um de seus elementos) e em caso de não serem, os uniam. A medida que ocorria um união de um grupo, a aresta que levou a unificação também era registrada em um array separado, quando esse array completava $n - 1$ elementos a iteração era então interrompida e a lista gerada era então a nossa árvore gerado mínima.

Como a árvore geradora mínima gerada pela função mantém sua ordenação, o que se foi feito em seguida para a determinação dos meios de transportes foi desconsiderar os trechos mais longos, pois estes seriam empregados os drones, e olhar nos restantes aqueles que se encaixavam nas características das motos. Após um iteração simples pelas arestas o valor final a ser pago

Algorithm 1 MergeSort

```
procedure MERGE(arestas, inicio, meio, fim)
    esquerda ← arestas[inicio : meio];
    direita ← arestas[meio+1 : fim];
    i, j ← 0;
    k ← inicio;
    while esquerda e direita não vazias do
        if distancia da aresta da esquerda é maior que a da direita then
            arestas_ordenadas[k] ← esquerda[i];
            i++, k++;
        if not then
            arestas_ordenadas[k] ← direita[j];
            j++, k++;
    while existir algum lado não vazio do
        arestas_ordenadas ← resto;
    arestas ← arestas_ordenadas
procedure MergeSort(clientes, inicio, fim)
    if inicio maior que fim then
        meio ← (meio+fim)/2;
        recursive ← (clientes, inicio, meio)
        recursive ← (clientes, meio+1, fim)
        MERGE ← (clientes, inicio, meio, fim)
```

Algorithm 2 KruskalAlgorithm

```
procedure Kruskal(vértices, arestas, conjuntos)
    arestas ← MergeSort(arestas);
    for e em arestas do
        if os vértices v,u são do mesmo conjunto then
            unifica os conjuntos de v,u;
            MST ← e;
    retorna MST
```

pela empresa foi então de fato obtido.

3 Complexidade

Considerando que existam m arestas no problema, teremos que o nosso algorítmo ao final tem um custo de $O(mlogm)$ (desconsiderando o custo de manipulação de arquivos).

Primeiramente temos as inserções dos dados. O custo mais alto da inserção é exatamente o da criação das arestas, que geram uma função de complexidade de $f(m) = m$ onde $m = n(n-1)/2$ que é relativo a dizer que o processo tem $O(m)$.

Após as inserções iniciais temos a ordenação. Nesse método, temos que a complexidade do algorítmo é exatemente $O(mlogm)$, como qualquer outro da família dos dividir para conquistar, com o bônus dele ter essa complexidade em qualquer caso e o ônus de ter essa complexidade também em espaço além de tempo.

Depois, ao realizar o Union-Find, observa-se que esse método também tem complexidade temporal de $O(mlogm)$. Isso pois, iteramos sobre as arestas checando se seus vértices são de conjuntos difrentes, e caso sejam, unificamos dois conjuntos pela inclusão do menor, o que representariá o log de m na ordem de complexidade, pois temos que os conjuntos a serem incluidos tendem de certa forma a seguir a mesma lógica na qual o MergeSort é concebido.

Por fim, temos a interação que ocorre no cáculo dos custos, mas como a iteração é limitada a $n-1$, temos que a complexidade desse trecho é bem menor que os outros devido ao quão grande o número de arestas m em um grafo conectado é maior que o de vértices n. E sendo assim, consderando a propriedade de soma de ordens de complexidades que permanece na solução a complexidade maior valor, temos que a complexidade final da solução é simplesmente $O(mlogm)$, tanto temporalmente quando espacialmente. Mas claro, isso desconsiderando o custo de buscar a memória no disco, pois se assim tivesse, teriamos que esse custo provavelmente se sobresairia sobre os outros.

4 Compilação

A solução deste problema foi totalmente implementado em C++, considerando como padrão sua versão 14, e devidamente compilado sem detectação de falhas ao rodar *make* em um Ubuntu 16.04.12 de g++ versão 5.4.0 20160609. Ao final da compilação será gerado um executável de nome *tp*02 e o mesmo pode ser utilizado com um arquivo de entrada tal como o seguinte exemplo:

$ *tp*01 *nome_do_arquivo_de_entrada.txt*

.