Trabalho Prático 1 - Grafos

André Lopes Gonzaga

3 de Junho de 2017

1 Introdução

Considere um site ou aplicativo de compartilhamento de rotas em que viajantes postam suas viagens marcadas para um futuro próximo. Ao postarem, eles anunciam sua disponibilidade como passageiros, motoristas ou ambos (caso um passageiro tenha carro e deixará que o sistema decida se vai utilizá-lo). O sistema então, irá agrupar esses viajantes e decidir quem será o motorista e os passageiros de cada viagem.

O único inconveniente para os passageiros é que eles deverão ir a pé da sua origem até a origem do motorista, esperar até o momento de partida do motorista, e depois, caminharem do destino do motorista até seu destino. Os motoristas, por outro lado, não terão de fazer desvios nem sofrerão com atrasos em sua rota, somente terão os assentos de seus automóveis ocupados por passageiros adicionais.

2 Definição do Problema

Essa variação do compartilhamento de rotas pode ser modelada utilizando teoria de grafos. Sendo $G = (V, A)$ um grafo direcionado e acíclico, onde $V = v_1, v_2, ..., v_n$ é um conjunto de viagens. Cada viagem v_i é uma tupla composta por posições e momentos de origem e destino, se é motorista, se é passageiro, número de passageiros e qual a capacidade do veículo. As arestas são pares direcionados (v_i, v_j) que conectam as viagens de passageiros v_i com viagens de motoristas v_j, desde que os passageiros de v_i possam partir de sua origem a pé e chegar a origem do motorista v_j antes da partida de v_j.

Cada aresta (v_i, v_j) possui um benefício associado $B(v_i, v_j) \in R^+$ que é dependente apenas de v_i, ou seja, todas as arestas que partem de v_i possuem o mesmo benefício. Nesse caso, $B(v_i, v_j)$ será igual à distância da origem ao destino de v_i. O benefício representa a distância que vi percorreria se ele não se juntasse a v_j.

O grafo G possui as possibilidades de compartilhamento, no entanto, é preciso selecionar aquelas relações que gerarão um maior benefício total para a rede. Sendo assim, a partir de um grafo $G = (V, A)$, deseja-se obter o plano de viagens de benefício máximo. Esse plano é um subgrafo de G dado por $G_p = (V, A_p)$ $A_p \subseteq A$ e deve satisfazer as seguintes restrições:

Restrição 1 $\forall(v_i, v_j) \in A_p$, não existe $k \neq j$ tal que $(v_i, v_k) \in A_p$. Essa restrição garante que uma rota vi não seja compartilhada com mais de uma rota, ou seja, que um passageiro não vá em mais de um automóvel ao mesmo tempo.

Restrição 2 $\forall(v_i, v_j) \in A_p$, não existe k tal que $(v_k, v_i) \in A_p$. Garante que uma rota vi só será compartilhada com v_j se não existir nenhuma outra rota v_k anteriormente compartilhada com v_i, ou seja, se uma rota for selecionada como de passageiro ela não poderá ser selecionada como motorista de outra viagem.

Restrição 3 O número total de passageiros que serão transportados por um veículo não pode exceder a capacidade do mesmo.

3 Solução Proposta

A solução proposta consiste em aplicar tentativa e erro nas diversidades possibilidades de achar benefício nas possíveis arestas do grafo. Dado um grafo $G = (V, E)$, a ideia proposta é testar as diversas combinações das arestas $(e_1, e_2, ..., e_n)$, sempre checando as condições de restrições dadas.

Logo, para uma entrada:

```
4
1 0 1 1 5 4.0
2 1 1 2 5 6.0
3 1 1 1 5 10.0
4 1 0 1 0 3.0
6
2 1
3 2
3 1
4 1
4 2
4 3
```

Temos seis tipos de arestas disponíveis e para tal, seguimos a seguinte lógica:

Algorithm 1 Fill set of edges

```
Initialize S = ∅
for each edge e ∈ E do
    S.add(e)
```

Então, após o algoritmo de preencher o consjunto S com todas as arestas possíveis do grafo, é necessário permutar todas as arestas deste conjunto para achar o maior benefício possível. Porém, um algoritmo de permutação é computacionalmente caro, de complexidade $n!$, logo abordaremos com detalhes na seção sobre complexidade.

Após o preenchimento do conjunto S, ao invés de permutar todas as combinações possíveis, nós selecionamos cada aresta e testamos as possíveis viagens possíveis de acordo com as suas devidas restrições. O algoritmo que segue realiza a lógica de permutação.

Algorithm 2 Permute and Calculate edges

```
Initialize best = ∅
maxBenefit = 0
for each edge e ∈ S do
    localBenefit = 0
    for each edge q ∈ (S − e) do
        checkAvailability(q)
        localBenefit += benefit(q)
    if maxBenefit < localBenefit then
        maxBenefit = localBenefit
        best = q ∈ maxBenefit
return maxBenefit
```

O que o algoritmo acima faz é checar para todos os tipos de combinação de arestas, se é possível adicionar uma aresta a_i e checa se o benefício é maior que o máximo benefício existente. Se for, o máximo benefício existente recebe o benefício calculado, e o conjunto de arestas deste benefício é adicionado a um conjunto chamado *best*.

O grande problema com a função *checkAvailability* é checar para um determinado número de arestas, quais são possíveis, ou seja, dada as três restrições existentes, se é possível adicionar a aresta local ou não. A primeira checagem é realizada no sentido de checar se o vértice de origem só pode ser passageiro, como mostra a função a seguir.

Algorithm 3 checkPassengerAvailability(Graph G, int vertexSource)

```
search for vertexSource in G
if vertexSource.Passenger is available then
    return true
return false
```

A segunda checagem é para verificar se há assentos disponíveis na viagem requerida. Já que cada vértice possui uma capacidade de assentos que é atualizada cada vez que adicionamos uma aresta, precisamos checar sempre antes de adicionar a aresta se é existem disponíveis.

A terceira e última checagem verifica se a aresta que está saindo de um determinado vértice é válida, ou seja, o vértice origem em questão não está recebendo nenhuma aresta (ou seja, não há passageiros chegando no determinado vértice), e uma vez que a aresta é criada, o vértice origem fica indisponível para receber arestas.

Uma vez que todas as checagens são realizadas e todas as opções de arestas testadas, temos o conjunto de arestas que formam o melhor benefício para as viagens e logo temos a solução do problema. Logo, a lógica geral da solução do problema é apresentada a seguir.

Algorithm 4 Solve Problem

1: Fill set of edges
2: best = Permute and Calculate edges
3: return **best**

4 Implementação

A linguagem escolhida para a implementação da solução proposta é **C**. Foram definidos dois tipos abstratos de dados, um representando o grafo em si e outro para representar e organizar a entrada e saída do programa.

4.1 Entrada e Saída

Para a estrutura de organização da entrada e saída, temos as 4 funções a explicadas a seguir.

4.1.1 readInput

Função responsável por ler a entrada de acordo com os comandos passados para o programa e fazer as checagens iniciais de parâmetros.

4.1.2 readNodes

Função responsável por ler todas as informações de vértices do arquivo passado e criar os vértices no grafo. A informação de como os vértices são criados e armazenados será mostrado na subseção de grafos mais a frente.

4.1.3 readEdges

Função que lê todos os dados de arestas do arquivo e liga o vértices de acordo com as informações do arquivo. A informação de como as arestas conectam os vértices do grafo estão apresentadas na subseção de grafos.

4.1.4 printCombination

Função que recebe o conjunto total de arestas e as combinações das mesmas e realiza as permutações necessárias checando as integridades. Além de chamar as funções de checagens de restrições, ela também calcula o melhor benefício e escreve a saída do problema.

4.2 Grafo

Para a estrutura do grafo foi escolhida a representação de listas de adjacência ao invés de matrizes pela sua facilidade de implementar na linguagem escolhida, como é mostrado na Figura 1. Cada vértice do grafo é armazenado na forma de lista encadeada e cada vértice que se conecta a ele é feito através de listas encadeadas saindo do próprio vértice. Ou seja, a estrutura implementada é uma lista de listas.

A estrutura do grafo em C é a seguir:

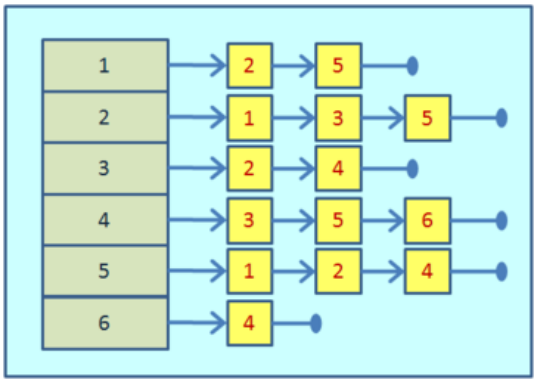

Figura 1: Lista de adjacência

```
typedef struct _ListAdj {
    int TravelId;
    struct _ListAdj *Next;
}ListAdj;

typedef struct _Graph {
    int TravelId;
    int Passenger;
    int Driver;
    int Amount;
    int Seats;
    float Benefit;
    int RemainingSeats;
    ListAdj *ListAdj;
    struct _Graph *Next;
}Graph;
```

4.2.1 createGraph

Recebe os diversos parâmetros que um vértice pode ter e cria a cabeça da lista de vértices, retornando o ponteiro para o mesmo.

4.2.2 createNode

Recebe os diversos parâmetros que um vértice pode ter, aloca o espaço de memória necessários, atribui todos os parâmetros ao novo vértice e retorna o ponteiro para o mesmo.

4.2.3 addNode

Adiciona um novo vértice ao grafo se conectando a lista encadeada principal.

4.2.4 createEdge

Cria um espaço de memória representando a aresta e retorna o ponteiro para a mesma.

4.2.5 addEdge

Cria, se possível, uma aresta saindo do vértice em questão. Caso não seja possível adicionar a aresta, por problemas de memória ou por já existir esta aresta, é apresentado uma mensagem de erro.

4.2.6 insertEdge

Insere uma aresta em um determinado vértice.

4.2.7 printGraph

Percorre todo o grafo imprimindo as informações necessárias de vértices e mostrando todas arestas conectadas aos vértices, ou seja, as possíveis viagens disponíveis.

4.2.8 calculateBenefit

Dado um vértice origem e um destino, checa se existe assentos disponíveis para a operação e se houver, aloca os assentos do vértice destino para o vértice origem e retorna o benefício da operação.

4.2.9 resetAvaiableSeats

Retorna todos as disponibilidades de assentos para o estado original.

4.2.10 checkPassengerAvailability

Checa se o vértice em questão é somente passageiro ou não.

5 Análise de Complexidade

Podemos dividir a análise de complexidade em três partes do algoritmo, que são: (a) criação do grafo; (b) permutação das arestas; (c) cálculo do máximo benefício.

5.1 Criação do Grafo

Para a criação do grafo, temos V linhas do arquivo de entrada que se transformarão em vértices, ou seja, $\mathbf{O(V)}$ para criação dos vértices. Para se conectar as arestas em cada um dos vértices, ou seja, ligar vértices com vértices, é necessário percorrer a lista de vértices disponíveis e inserir na posição certa.

Utilizando da análise amortizada, e assumindo que o arquivo de entrada está sempre ordenado, temos a ordem de complexidade de $\mathbf{O(V + E)}$ para a criação do grafo com as listas de adjacência.

5.2 Permutação das Arestas

Dado um vetor de n posições, uma simples permutação em seu elementos tem complexidade igual a $\mathbf{O(n!)}$, o que é inconcebível para resolver problemas um pouco maiores. Para evitar o problema de permutar todos os elementos, combinamos somente os elementos possíveis e que precisamos realizar as checagens de compatibilidade com informações do grafo.

Por exemplo, para a entrada:

```
3
1 0 1 1 5 4.0
2 1 1 2 5 6.0
3 1 1 2 5 10.0
2
2 1
2 3
3 1
```

Ao selecionar o par $(2, 1)$, não faz sentido selecionar $(2, 3)$, pois o elemento 2 nunca poderá ter dois destinos. Logo, automaticamente descartamos o par $(2, 3)$. Ao selecionar $(2, 3)$, a mesma regra se aplica para $(2, 1)$, onde excluímos essa combinação e prosseguimos em frente.

Desta forma, reduzimos consideralmente o tempo de execução do algoritmo a fim de ter um tempo viável. Logo, no pior caso o algoritmo fará a checagem de todos os elementos duas vezes a fim de certificar que não está checando ninguém por engano. Então, a ordem de complexidade desta parte é de $\mathbf{O(V \times V)}$.

Porém, além da complexidade das combinações, temos a complexidade associada a percorrer o grafo em busca das restrições para cada combinação selecionada. Caso a combinação seja possível, incluímos a mesma ao resultado e adicionamos o benefício local ao conjunto de benefícios. Para

percorrer o grafo e encontrar as restrições no mesmo, temos a mesma complexidade de criação do grafo que é $\mathbf{O(V + E)}$.

Como as checagens de restrições estão dentro do *loop* das combinações, temos uma complexidade final de: $\mathbf{O(\ (V \times V) \times (V + E)\ )}$, que é quase infinitamente melhor que $\mathbf{O(n!)}$.

5.3 Cálculo do máximo benefício

Para o cálculo do máximo benefício é necessário checar todos os benefícios encontrados e escolher o melhor. A complexidade de realizar tal operação, dado que os benefícios são checados de acordo com sua entrada na lista de possíveis soluções, é $\mathbf{O(1)}$ pois precisamos somente comparar com o melhor benefício no momento e se o benefício calculado for maior, o algoritmo substitui com o novo valor.

6 Conclusão

Neste trabalho foi mostrado o quão difícil é trabalhar com problemas utilizando grafos. Mesmo com uma boa implementação de suas estruturas, os algoritmos tendem a ter complexidade exponenciais e é necessário que os cientistas da computação intervenham para não se ter custos computacionais astrônomicos.

Para um simples problema de compartilhamento de viagens, conseguimos reduzir a complexidade de tentativa e erro de $\mathbf{O(n!)}$ para $\mathbf{O(\ (V \times V) \times (V + E)\ )}$, mas que ainda é uma complexidade grande já que se um grafo tiver muitas arestas (chegando quase ao número de vértices), podemos ter a complexidade $\mathbf{O}(V^4)$.

Apesar da decisão de implementar o problema em uma linguagem que muitos desenvolvedores fogem, a linguagem C consegue facilitar a implementação e as estruturas acabam ficando mais claras e intuitivas de seguir. A decisão da linguagem foi acertada neste quesito.

Infelizmente, devido ao tempo, para algumas entradas específicas o algoritmo não retorna o maior benefício esperado. A explicação está no algoritmo de permutação que analisa o conjunto de arestas de forma linear e para algumas entradas específicas (onde o benefício está de forma não linear), o benefício retornando não é o máximo, mas é perto do máximo.

Outro ponto a ser melhorado é a desalocação de memória quando o algoritmo termina. É necessário que após todo o uso da memória alocada para a construção dos vértices e das suas listas de adjacência, o próprio programa percorra suas estruturas criadas e dê um `free()` para a liberação de memória. Mas novamente devido ao tempo, não foi possível a implementação.