Trabalho Prático 2 - Paradigmas de Programação

André Lopes Gonzaga

2 de Julho 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.

Para este trabalho foram desenvolvidos três algoritmos para a resolução do problema, que envolvem três diferentes paradigmas de programação. O primeiro é um algoritmo que usa **força bruta** para a resolução, o segundo usa da técnica **gulosa** de otimizações locais e a última **programação dinâmica**, que faz a sobreposição de problemas e define uma subestrutura ótima de subproblemas. Os algoritmos serem descritos com mais detalhes na Seção 3.

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.

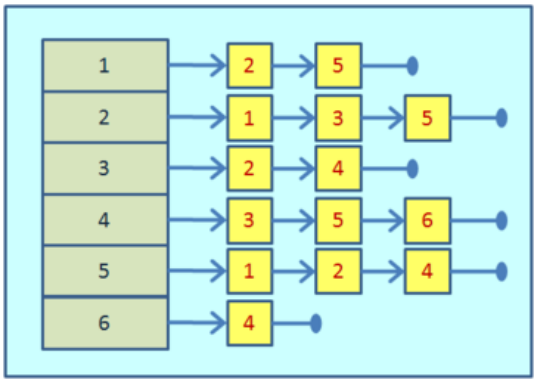

Figura 1: Lista de Adjacência

3 Implementação

A linguagem escolhida para a implementação foi **C**. Foram definidos três tipos abstratos de dados, um representando o grafo em si, outro para representar e organizar a entrada e saída do programa e um último para representar uma lista de listas.

3.1 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 foi implementado segundo as definições abaixo:

```
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;
    int Available;
    int isDriving;
    float benefitIncoming;
    ListAdj *ListAdj;
    struct _Graph *Next;
}Graph;
```

3.2 Entrada e Saída

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

3.2.1 readInput

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

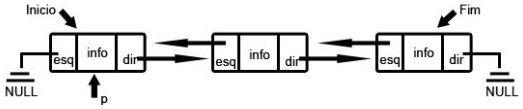

Figura 2: Lista Duplamente Encadeada

3.2.2 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.

3.2.3 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.

3.3 Lista

Uma lista é definida como um sequência de elementos $< a_1, a_2, ..., a_n >$ onde para cada elemento existe um ponteiro para o próximo, quando houver. Com este tipo abstrato é possível representar um conjunto de elementos em uma determinada ordem, e as possíveis operações de inserção, remoção, visualização e edição são permitidas.

Para cumprir um dos objetivos deste trabalho, foi usado uma lista de listas, onde para cada elemento a_i da lista principal, além do ponteiro para o próximo elemento da lista, há um ponteiro para uma lista adjacente. Esta representação é bem parecida com a representação do grafo, mostrado na Figura 1.

Contudo, houve a necessidade de implementar uma lista duplamente encadeada para as listas adjacentes, onde para cada elemento a_i há um ponteiro para o próximo elemento e para o antecessor. Também, além do ponteiro partindo da lista principal para o começo da lista de adjacência, há também um ponteiro para o último elemento da lista. Com isso, é possível perorrer a lista nos dois sentidos, tanto de frente para trás como de trás para frente. Um exemplo de lista encadeada é mostrado na Figura 2.

A implementação da lista foi feita como mostrado a seguir:

```
typedef struct _ListListAdj {
    int From;
    int To;
    float Benefit;
    struct _ListListAdj *Next;
    struct _ListListAdj *Prev;
}ListListAdj;

typedef struct _List {
    int From;
    int To;
    float Benefit;
    ListListAdj *ListAdj;
    ListListAdj *End;
    struct _List *Next;
}List;
```

4 Algoritmos

Esta seção apresenta os diversos tipos de paradigmas implementados neste trabalho.

4.1 Bruto

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.

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.

Algorithm 3 checkPassengerAvailability(Graph G, int vertexSource)

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

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

```
Fill set of edges
best = Permute and Calculate edges
return best
```

4.2 Algoritmo Guloso

A solução proposta pelo algoritmo guloso consiste em realizar decisões locais a fim de achar uma solução ótima no final. 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,
```

são feitas duas ordenações: pelos vértices com maiores benefícios de saída e pelos vértices com maiores valores de benefícios de entrada. Para o caso desta entrada, os valores ordenados por saída de benefício é:

```
3 10.0
2 6.0
1 4.0
4 3.0,
```

e os valores ordenados pelos benefícios de entrada é:

```
1 19.0
2 13.0
3 3.0
4 0.0.
```

Uma vez ordenados os vértices, temos duas listas que possamos explorá-las para obter uma solução. Para isso, percorremos a lista de vértices com maior benefício de entrada e procuramos qual é o vértice com maior benefício que se conecta a ele. Assumimos que esta associação é a melhor solução que podemos fazer, já que não temos como controlar todas as probabilidades de associação em tempo viável.

Ou seja, a solução gulosa assume que conectando o vértice que tem maior benefício de saída e o vértice que tem o maior benefício de entrada vai representar a melhor escolha local. O pseudo-código abaixo demonstra como a estratégia gulosa se comporta em termos de código.

Algorithm 5 Greedy Algorithm

```
1: Sort vertex by incoming benefit
2: Sort vertex by outgoing benefit
3: for u ∈ sorted list by incoming benefit do
4:     for v ∈ sorted list by outgoing benefit do
5:         CheckRestrictions(v, u);
6:         if restrictions are good then
7:             AddTravel(u, v);
8: return best
```

Porém, este algoritmo não obtém a solução ótima para todos os casos. Intuitivamente, podemos pensar que sempre ao considerar que o vértice com maior benefício de entrada vai se conectar ao vértice com maior saída, um conjunto de vértices intermediários pode estar sendo excluído de se conectar ao vértice com maior saída, o que representaria um maior benefício global.

Apesar de não chegar na solução ótima em todos casos, o algoritmo guloso possui uma ordem de complexidade menor que os demais, resultando em um tempo de execução muito menor que o algoritmo bruto e com um relativo ganho para a programação dinâmica.

4.3 Programação Dinâmica

Para a programação dinâmica, onde a substrutura ótima e a sobreposição de subproblemas devem ser implementados, foi necessária a implementação de uma estrutura auxiliar ao grafo. Esta estrutura é a lista de listas duplamente encadeadas e Durante a explicação da programação dinâmica será discutido o porque da utilização desta estrutura.

Como no algoritmo guloso, há uma ordenação pelos vértices que possuem o maior benefício, e adicionamos os mesmos na lista criada. Nesta altura já temos o grafo criado também e consequentemente sabemos para quais vértices as arestas apontam. Vamos utilizar a mesma entrada de exemplo do algoritmo bruto e do guloso para explicação.

```
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
```

Da mesma forma que o algoritmo guloso, a ordenação de vértices por benefícios é:

```
3 10.0
2 6.0
1 4.0
4 3.0
```

Como 3 é o que tem maior benefício, colocamos ele na lista com os vértices que ele aponta como demonstrado na Figura 3. Uma vez conectados os vértices que 3 aponta, passamos para o próximo item da lista ordenada e adicionamos nas listas adjacentes (que são duplamente encadeadas), como sugere a Figura 4

O próximo item da lista é o vértice 1 que tem peso 4, mas que não possui nenhuma aresta saindo dele. Logo, passamos para o último vértice que é o 4, que possui benefício 3.0. Porém,

Figura 3: Todas as arestas do vértice 3 representadas na lista

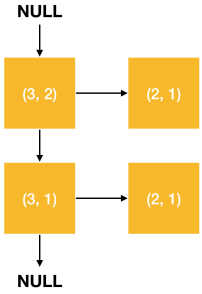

Figura 4: Todas as arestas do vértice 2 representadas na lista de adjacência

este vértice possui várias arestas saindo dele, que são para 1, 2 e 3. Logo, a lista de completa das combinações usando programação dinâmica é representado na Figura 5.

O pseudo-código do algoritmo é mostrado a seguir.

Algorithm 6 Dynamic Programming

```
1: Sort vertex by benefit
2: for v ∈ sorted list by incoming benefit do
3:     for v ∈ sorted list by outgoing benefit do
4:         CheckRestrictions(v, u);
5:         if restrictions are good then
6:             AddTravel(u, v);
7: return best
```

4.4 Substrutura Ótima

A garantia da substrutura ótima está no fato do algoritmo sempre considerar os vértices com maior benefício, e assim nunca ficar para trás em termos de benefício global. Para cada iteração, onde vértices são adicionados na lista, a subestrutura ótima é montada com cada inserção, e no final há a certeza da solução ótima.

Uma grande diferença para o algoritmo guloso é o fato de considerar não só os vértices com maior benefício, mas também adicionar um vértice coringa com benefício zero. Em alguns cenários o maior benefício global não inclui o vértice de maior benefício e adicionando sempre um vértice coringa garante a otimalidade da solução.

4.5 Sobreposição de Subproblemas

Para a sobreposição de subproblemas, ou seja, para evitar de calcular sempre os benefícios de subconjuntos já conhecidos, o algoritmo percorre a lista já criada e checa a existência do benefício já calculado. Se este já estiver calculado, não precisamos recalcular e só retornamos o benefício existente. Caso não esteja calculado, fazemos o cálculo e adicionamos na lista para ser usado no futuro.

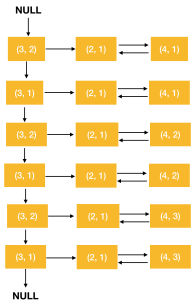

Figura 5: Todas as arestas do vértice 4 representadas na lista de adjacência

Desta forma, assim como na solução de programação dinâmica para o problema da sequência de Fibonacci, não precisamos recalcular toda vez que o problema se deparar com um benefício para calcular. Para a complexidade de tempo percebe-se que há uma redução do seu custo, mas como precisamos armazenar em uma lista os benefícios já calculados temos uma ordem de complexidade de espaço maior.

5 Ordem de Complexidade

Em termos de ordem de complexidade, vamos analisar não só a complexidade por tempo mas também o quanto cada algoritmo consome de espaço. Como cada algoritmo já foi explicado em detalhes na Seção 4, vamos nos limitar exclusivamente as análises de complexidade. A Tabela 1 mostra as diversas ordens de complexidade.

5.1 Algoritmo Bruto

O algoritmo bruto tenta todas as combinações possíveis, entre diversas arestas e vértices. Logo, é fácil provar que para uma grafo de n vértices e m arestas, a complexidade de se computar todas as combinações é $O((m)!/n!(m-n)!)$.

Para a complexidade de espaço, se consome o necessário para o armazenamento do grafo e a matriz para a computação das possíveis soluções. Como é sabido que a complexidade de um grafo pode ser representada por $O(m+n)$, e uma matriz se tem a ordem de n^2, onde n é o número de vértices do grafo, temos de fato uma ordem de complexidade de espaço no pior caso de $O(n^2)$.

5.2 Algoritmo Guloso

Para o algoritmo guloso, vamos considerar somente a parte mais complexa do código onde os loops e computações se concentram. Então para cada n vértices, é calculado qual tem o maior em ordem linear $O(n)$. Para cada vértice calculado como maior benefício, é procurado qual o vértice tem maior benefício de entrada, o que também tem custo linear de $O(n)$, mas como estamos computando para cada n, então ambas as computações tem o custo de $O(n^2)$.

Além disso, após a computações dos vértices que se conectam, precisamos percorrer a lista de combinações de arestas que restam dentro de um vetor de $n-2$ elementos, o que corresponde ao custo de $O(n)$. Então, para todo o cálculo de combinações do algoritmo guloso temos um custo de tempo de $O(n^3)$ somado ao fato de percorrer o grafo em busca do seu benefício que também tem custo de no máximo $O(n)$, e logo o custo total de tempo do algoritmo guloso é $O(n^4)$.

Para o custo de espaço, temos além do custo de armazenar o grafo, o custo de armazenar 2 matrizes que auxiliam no cálculo das combinações da solução gulosa. Logo, no pior caso temos o custo da matriz, que é $O(n^2)$.

5.3 Programação Dinâmica

Na programação dinâmica temos o mesmo custo de tempo do grafo $O(n+m)$. Uma vez que há a ordenação dos n elementos (foi usado o método de ordenação seleção, com ordem de complexidade $O(n^2)$ foi feita, é criada a lista para armazenar os subproblemas já calculados e também para armazenar a subestrutura ótima.

Complexidade	Bruto	Guloso	Dinâmico
Tempo	$O((m)!/n!(m-n)!)$	$O(n^4)$	$O(n!(m-n)!)$
Espaço	$O(n^2)$	$O(n^2)$	$O(n^2)$

Tabela 1: Comparação de ordem de complexidade entre os algoritmos

Entradas	Bruto	Guloso	Dinâmico
toy_0.in	0.000472	0.000198	0.000201
toy_1.in	0.000489	0.000219	0.000216
toy_2.in	0.001090	0.000218	0.000337
toy_3.in	0.000792	0.000205	0.000312
toy_4.in	0.001151	0.000212	0.000232
toy_5.in	0.000483	0.000187	0.000314
toy_6.in	0.023087	0.000278	0.000359
toy_7.in	0.000881	0.000225	0.000187
toy_8.in	0.001365	0.000209	0.000318

Tabela 2: Comparação de tempo entre os algoritmos

Então, em n vezes, é copiada a lista anterior, em ordem de tempo $O(n)$ e se computa as possíveis próximas combinações de vértices, que é $(n!(m-n)!$ e que resulta em um total de complexidade de $O(n!(m-n)!)$ no pior caso, o que é relativamente melhor que o bruto pois há podas na árvore de soluções.

Para a complexidade de espaço, além do custo do grafo $O(n+m)$ temos também o custo da lista que pode ter no pior caso $O(n^2)$, que é o custo total de espaço da programação dinâmica.

6 Resultados

Para efeito de comparação entre os três algoritmos implementados, foram medidos os tempos para cada uma das entradas *toy* fornecidas pelo problema. Os resultados podem ser vistos na Tabela 2. Percebe-se, como era de se esperar pelas ordens de complexidade de tempo, que o guloso deveria gastar menos tempo que o bruto e que ele tem uma leve vantagem sobre o dinâmico.

Para entradas muito grandes, somente o guloso consegue resolver em tempo hábil, já que para o Bruto e o Dinâmico os custos impossibilitam a resolução. Também o fato do uso de muitos ponteiros fazem com que a resolução de problemas um pouco maiores fique pesado em termos de manipulação de memória e pode ser que o algoritmo não retorne uma saída.

Em alguns cenários, se percebe uma leve vantagem do algoritmo de programação dinâmica, e a razão é pelo fato da otimização dos cálculos de benefício. Apesar do guloso armazenar o benefício para cada aresta no grafo, não há uma sobreposição de subproblemas. Todas as operações de cálculo de benefício envolve percorrer o grafo, o que o torna um pouco mais pesado quando existe uma pequena quantidade de vértices.

Os testes foram todos rodados em um MacBook 2015, processador 2.7 GHz Intel Core i5 e memória RAM 16 GB 1867 MHz DDR3.

7 Conclusão

Neste trabalho foi mostrado o quão difícil é trabalhar com problemas utilizando grafos, e usamos três diferentes paradigmas de programação. 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.

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.

Ficou claro neste trabalho que o algoritmo guloso quando não consegue chegar na solução ótima para alguns cenários, estabelece um $trafe-off$ com um algoritmo mais eficiente. Já com a programação dinâmica, garantimos a otimalidade das subestruturas e por causa da sobreposição de subproblemas, ganhamos em perfomance não abrindo mão da qualidade da solução.

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.