Projeto e Análise de Algoritmos: Trabalho Prático 1

João Mateus de Freitas Veneroso

Departamento de Ciência da Computação da Universidade Federal de Minas Gerais

June 14, 2017

Introdução

Este relatório descreve a implementação do trabalho prático 2 da disciplina Projeto e Análise de Algoritmos. O trabalho consistiu em resolver um problema de compartilhamento de viagens por meio de três paradigmas diferentes: força bruta, programação dinâmica e algoritmos gulosos.

Força Bruta

O algoritmo de força bruta descrito abaixo é o mesmo implementado no Trabalho Prático 1.

O algoritmo 1 implementa uma solução de força bruta para o problema de compartilhamento de viagens. Primeiro inicializa-se o benefício máximo b^* com um valor negativo, dessa forma qualquer configuração válida vai proporcionar um benefício maior. A partir daí, no loop das linhas 5-10, para cada configuração válida, calcula-se o benefício total e atualiza-se o valor de b^* se este benefício for maior do que qualquer um encontrado até então. Além disso, a variável G_p^* guarda a configuração que proporcionou o maior benefício. Ao final do algoritmo, b^* guarda o valor do benefício máximo para o grafo G e G_p^* guarda a configuração que proporcionou este benefício.

A complexidade deste algoritmo depende do número de configurações G_p diferentes e do custo da função $ConstraintsAreValid$. O número de configurações G_p diferentes é 2^m para m igual ao número de arestas no grafo G. Pois, cada aresta pode estar presente ou não em G_p e nós queremos as combinações possíveis para todas as arestas m. O custo da função $ConstraintsAreValid$ depende do número de arestas, pois, para cada aresta (u, v), temos de verificar se ela é a única aresta que sai do vértice u, se o vértice v é um motorista e se v

possui espaço para acomodar todos os passageiros de u. Como todas estas operações tem custo constante, a função $ConstraintsAreValid$ tem custo $O(m)$. Por último, a linha 7 calcula a soma dos pesos de todas as arestas também com custo $O(m)$. Logo, a complexidade total do algoritmo é $2m2^m$ e o algoritmo é $O(m2^m)$.

Algorithm 1 MaximizeBenefit

1: **procedure** MAXIMIZEBENEFIT(G(V,A))
2:     $b^* \leftarrow -1$
3:     $G_p^* \leftarrow \emptyset$
4:
5:     **for all** $G_p = (V, A_p) : A_p \subseteq G.A$ **do**
6:         **if** *ConstraintsAreValid(G_p)* **then**
7:             $b \leftarrow \sum_{(v_i,v_j)\in A_p} B(v_i, v_j)$
8:             **if** $b > b^*$ **then**
9:                 $b^* \leftarrow b$
10:                 $G_p^* \leftarrow G_p$

Uma melhora ainda pode ser alcançada se deixarmos de testar parte das configurações inválidas de G. Sabemos que um passageiro somente pode pegar carona em um carro, portanto, qualquer combinação G_p onde existirem arestas $(u, v_i), (u, v_j) : i \neq j$ é inválida. Dessa forma, basta manter uma única aresta ativa por vez na lista de adjacência de cada vértice. É necessário lembrar que um vértice pode representar um motorista sem passageiros ou um grupo de passageiros que não vai pegar carona com ninguém, portanto, o número total de combinações se torna $\sum_{v\in V} G.Adj(v) + 2$ e, no pior caso (um grafo completo), o número de combinações se torna $O(n^n)$ onde n é o número de vértices no grafo G. Feitas estas considerações, a complexidade assintótica do algoritmo é $O(mn^n)$. Apesar da complexidade parecer pior do que o algoritmo descrito anteriormente, devido a forma como o algoritmo é construído ele é sempre necessariamente melhor ou igual ao algoritmo anterior. Pois, no máximo, ele testa todas as combinações de arestas 2^m. O algoritmo de força bruta implementado em Python para este trabalho utiliza este método.

A complexidade de espaço do algoritmo original e da versão aprimorada é $O(n + m)$, pois o grafo é armazenado na forma de listas de adjacência.

Programação Dinâmica

Em qualquer solução válida para o problema dos compartilhamentos de viagem, os passageiros-motoristas tem de decidir se vão atuar exclusivamente como passageiros ou motoristas devido às restrições do problema. Dessa forma, sendo k o número de passageiros-motoristas, ao variar quais deles atuam como passageiros e quais atuam como motoristas, existem 2^k combinações possíveis que dão origem a problemas distintos. Cada um destes problemas pode ser pensado

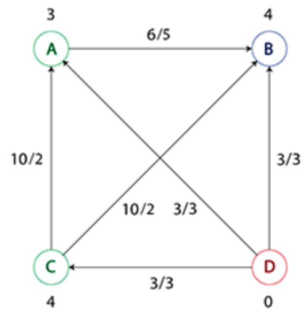

Figure 1: Exemplo do problema de compartilhamento de viagens modelado em forma de grafo

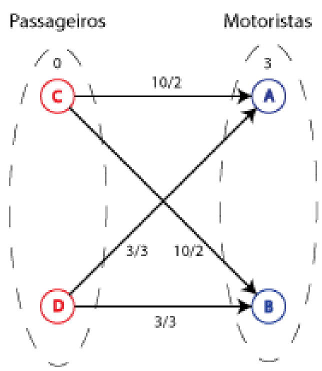

Figure 2: Grafo bipartido após definição dos passageiros e motoristas no grafo da figura 1

como um grafo bipartido G(V,A) onde os vértices de passageiros pertencem a um subconjunto de V e os vértices de motoristas pertencem a um outro subconjunto de V, sendo que todas as arestas conectam um vértice passageiro a um vértice motorista e os dois subconjuntos são disjuntos.

A figura 1 mostra um exemplo do probema de compartilhamento de viagens modelado na forma de um grafo, onde vértices vermelhos representam grupos de passageiros, vértices azuis representam motoristas e vértices verdes representam passageiros-motoristas. Os números próximos aos vértices representam a capacidade do veículo e os números próximos às arestas (v_i, v_j) representam o benefício do compartilhamento $B(v_i, v_j)$ e o número de passageiros na viagem $Q(v_i, v_j)$ separados por uma barra. Perceba que os vértices A e C são passageiros-motoristas e os demais vértices são passageiros ou motoristas exclusivos. Ao selecionar o vértice A como motorista e o vértice C como passageiro, podemos modelar o problema com o grafo bipartido da figura 2. Para cada uma das combinações de passageiros e motoristas podemos construir um grafo bipartido similar. Nesta forma, o problema passa a se parecer bastante com o *0-1 Multiple Knapsack Problem*, com a diferença que alguns items podem estar restritos à sacolas específicas. No caso, as sacolas representam os carros e os itens representam os grupos de passageiros. Além disso, é possível mapear o *0-1 Multiple Knapsack Problem* no problema de compartilhamento de viagens se considerarmos o caso em que todos os passageiros podem pegar carona em todos os veículos. Logo, o problema do compartilhamento de viagens é NP-difícil.

Dado um grafo $G(V, A)$, cada motorista v_j dá origem a um *0-1 Knapsack Problem*. Pois, cada aresta $(v_i, v_j) \in G.A$ representa uma possbilidade do grupo de passageiros v_i seguir viagem com v_j. Logo, para maximizar o benefício dos compartilhamentos do motorista v_j ($B_{max}(G, v_j)$) é preciso escolher o conjunto de arestas (v_i, v_j) que confira o maior benefício sem exceder a capacidade do carro, que é basicamente a definição do *0-1 Knapsack Problem*. No entanto, a forma como um motorista maximiza o benefício dos seus compartilhamentos influencia como os outros motoristas podem maximizar os seus compartilhamentos. Logo, assumindo que o motorista v_j está levando o grupo de passageiros X, o próximo motorista deverá maximizar seu benefício com base no grafo $G'(V - X - v_j, A')$, sendo que A' é o conjunto de arestas original A sem as arestas que contém v_j ou qualquer um dos passageiros levados por ele. No caso de motoristas que podem ser passageiros, também existe a possibilidade do motorista v_j não levar passageiro nenhum. Nesse caso, as arestas incidentes em v_j serão apagadas do grafo G, mas as arestas que saem de v_j serão mantidas e, o vértice v_j também será mantido.

Se o motorista v_j decide levar um passageiro v_i, os demais motoristas vão maximizar seus benefícios sem considerar o passageiro v_i no grafo. Por conta disso, o benefício encontrado não será necessariamente máximo. Por exemplo, tome o caso em que v_j pode levar um dos grupos de passageiros $\{p_1 = 12/2, p_2 = 20/1, p3 = 1/1\}$ onde o primeiro valor associado a cada grupo representa o seu benefício e o segundo valor representa o número de pessoas naquele grupo (note que a barra não representa divisão nesse caso). Assumindo que $v_j.capacidade = 2$, v_j maximiza o benefício dos seus compartilhamentos se escolher os passageiros $\{p_2, p_3\}$ auferindo um benefício total de 21. Agora assuma que existe um outro motorista v_{j+1} que pode compartilhar viagem apenas com p_2. Nesse caso, o motorista v_{j+1} ficará impedido de compartilhar essa viagem. No entanto, o benefício máximo dos compartilhamentos seria claramente maior se v_j tivesse escolhido levar $\{p_1\}$ ao invés de $\{p_2, p_3\}$.

Por conta da incapacidade de prever localmente qual é o benefício total máximo do plano de compartilhamento, é necessário testar cada uma das combinações de k passageiros para o motorista v_j e escolher aquela que provê o benefício total máximo. Isto dá um total de 2^k combinações, onde, para cada combinação de passageiros, constrói-se um grafo G' sem aqueles passageiros e sem o motorista ou apenas sem as arestas dos passageiros se o motorista não for levar ninguém, como explicado anteriormente. Então, calculamos o benefício máximo para o subgrafo G' e assim por diante. Sendo que cada combinação de passageiros em cada motorista pode ter seu benefício máximo calculado pela resolução do *0-1 Knapsack Problem.*

Finalmente, podemos construir a seguinte equação de recorrência:

$$B_{max}(G, D_j) = \begin{cases} 0, & \textit{se } G = \emptyset. \\ \max\limits_{P_k \subseteq D_j.P} B_{max}(G', D_{j-1}) + B(P_k), & \textit{caso contrário.} \end{cases} \tag{1}$$

onde P_k são todos as combinações de passageiros do motorista D_j, onde $P_k \subseteq D_j.passageiros \subseteq G.A$. G' depende da escolha dos passageiros como explicado nos parágrafos anteriores e $B(P_k)$ é o somatório dos benefícios dos passageiros em P_k. A ordem de escolha dos motoristas D_j não é importante uma vez que todas as possíveis combinações de passageiros acabam sendo testadas de uma forma ou de outra. No entanto, esta ordem será importante quando tratarmos da solução gulosa para o problema.

Perceba que para calcularmos o benefício máximo de G é necessário resolver maximizar os benefícios nos subproblemas G'. Ou seja, o problema tem subestrutura ótima. A prova deste postulado é que, se obtemos o benefício máximo em G com base em um $B(G')$ cujo benefício não é máximo, então, podemos simplesmente substituir $B(G')$ por $B_{max}(G')$ e obter um benefício maior, contradizendo o fato de G ser máximo inicialmente. Portanto, $B(G')$ tem de ser máximo.

A implementação ingênua desta equação de recorrência acaba calculando várias vezes o mesmo problema, pois, um subgrafo G' pode ocorrer diversas vezes na árvore de recursão, mas o seu benefício máximo $B_max(G')$ é sempre o mesmo. Portanto, podemos ganhar eficiência ao salvar os resultados dos cálculos anteriores do *0-1 Knapsack Problem.* Além disso, podemos utilizar uma abordagem de programação dinâmica para resolver o próprio *0-1 Knapsack Problem.*

O *0-1 Knapsack Problem* pode ser resolvido de forma simples gerando uma tabela onde as linhas representam os passageiros $\{p_1, p_2, ..., p_n\} \in D_j.P$ e as colunas representam a capacidade de D_j de 1 até $D_j.capacidade$. Então, calculamos para cada linha, o benefício máximo dado o limite imposto pelas colunas. Na próxima linha, verificamos se podemos incluir o próximo passageiro para cada uma das capacidades e, se isso não for possível, repetimos o valor da célula de cima. Se for possível incluir o passageiro, então verificamos o valor da linha de cima com a capacidade da coluna atual menos o número de passageiros adicionados. Devido a uma limitação de espaço este algoritmo não será explicado em mais detalhes. No entanto, o código implementa ele conforme os passos aqui descritos.

No caso do cálculo do benefício máximo, podemos aplicar uma abordagem de programação dinâmica *Bottom Up.* Digamos que $D = D_1, D_2, ..., D_n$ é a lista dos motoristas no grafo G. E, o grafo G possui 2^n subgrafos com combinações diferentes dos vértices $G.V$, ao se retirar os vértices um a um em todas as combinações possíveis. Consideremos agora o problema de encontrar o benefício máximo para cada uma das combinações de $G.V$ considerando apenas os motoristas

$\{D_1, D_2, ..., D_i\}$. Como demonstrado anteriormente, este problema tem subestrutura ótima e as construções para $D_{i+1}, D_{i+2}, ...$ dependem de sua solução. Podemos produzir uma matriz B_{max} com n linhas representando os conjuntos de motoristas e 2^n colunas representando todas as possíveis combinações $V_j \subseteq G.V$, sendo que os índices são a codificação binária das combinações. Por exemplo, $\{0 \Rightarrow \emptyset, 1 \Rightarrow \{V_1\}, 2 \Rightarrow \{V_2\}, 3 \Rightarrow \{V_1, V_2\}, ...\}$. Cada célula $B_{max}[i][j]$ representa o benefício máximo considerando apenas os motoristas $\{D_1, D_2, ..., D_i\}$ no grafo G' onde $G'.v_k \in V_j \forall G'.v_k \in G'.V$. O tamanho total da matriz B_{max} será $O(n2^n)$ sendo n o número de vértices no grafo G.

Algorithm 2 DPMaximizeBenefit

```
procedure DPMaximizeBenefit(G(V,E))
    Drivers := GetDrivers(G)
    B_max := Array[Drivers.length]
    Combinations := |G.V|^2
    i := 0

    for all i < Drivers.length do
        B_max[i] := Array[Combinations]

    for all D_i ∈ Drivers do
        for all V_j ⊆ G.V do
            if D_i ∈ V_j then
                Benefit := 0
                Solution := 0-1 Knapsack(D_i, V_j)
                for all v_k ∈ Solution do
                    Benefit := Benefit + B(v_k, D_i)
                    G.V := G.V − {v_k}
                if Solution ≠ ∅ then
                    G.V := G.V − {D_i}
                CombinationIndex := GetCombinationIndex(G.V)
                Benefit := Benefit + B_max[i − 1][CombinationIndex]
                Replace all removed vertices in graph G
                B_max[D_j][P_comb] = Benefit
            else
                B_max[i][j] = B_max[i − 1][j]
    MaxBenefit := 0
    i := Drivers.length − 1

    for all V_j ⊆ G.V do
        if B_max[i][j] > MaxBenefit then
            MaxBenefit := B_max[i][j]
```

Finalmente, o algoritmo para a abordagem *Bottom Up* está descrito em

??. Os nomes de todas as variáveis são equivalentes aos definidos nos últimos parágrafos. A função *0-1 Knapsack*(D_i, V_j) resolve o *0-1 Knapsack Problem* sendo D_i a sacola com capacidade D_i*.capacidade* e sendo V_j os itens que podem ser colocados na sacola, atentando para o fato que só são considerados os vértices de V_j que podem ser passageiros de D_i. A função *GetCombinationIndex(G.V)* simplesmente retorna o índice j equivalente a combinação de vértices em $G.V$. A complexidade deste algoritmo é $O(n2^n O(0-1 KnapsackProblem(n))$. Não é uma solução muito eficiente, mas devido ao fato do algoritmo ser NP-difícil, todas as soluções serão de ordem exponencial.

Algoritmo Guloso

Nesta seção será descrito um algoritmo que utiliza uma estratégia gulosa para escolher a ordem dos motoristas no algoritmo da seção anterior e que também utiliza uma estratégia gulosa para resolver o *0-1 Knapsack Problem*. Começemos pela resolução guloso do *0-1 Knapsack Problem*.

Os itens da sacola (passageiros) serão ordenados por ordem não crescente de eficiência, sendo que eficiência é definida por:

$$\frac{benefcio}{nmerodepassageirosnaviagem}$$

a estratégia gulosa vai escolher os itens nesta ordem até que o carro não comporte mais nenhum passageiro. Adicionalmente, conferimos se colocar apenas o passageiro de maior benefício dentro do carro vai conferir um benefício maior do que a estratégia descrita e, se este for o caso, substituímos os passageiros escolhidos pelo passageiro de maior benefício. Esta estratégoa não garante uma resolução ótima para o problema, mas é bastante rápida com complexidade $O(nlogn)$.

Agora, dado um grafo G, escolheremos o motorista cujo benefício calculado pelo *0-1 Knapsack Problem* é o maior entre todos os motoristas do grafo. Então, adicionamos o benefício obtido ao benefício total e retiramos tanto o motorista quanto todos os passageiros escolhidos e suas arestas do grafo G, produzindo o grafo G'. Repetimos o processo até que não haja mais motoristas no grafo e então obtemos o benefício total. Da mesma forma que o *0-1 Knapsack Problem*, esta estratégia não produz necessariamente um resultado ótimo, mas a sua complexidade é $O(n^3 logn)$ onde n é o número de vértices no grafo. O que é consideravelmente melhor do que a estratégia de força bruta ou a abordagem de programação dinâmica.

A abordagem gulosa para o problema da maximização de benefício no grafo de compartilhamento de viagens está descrita no algoritmo **??**. A função *KnapsackGreedy* utiliza a estratégia gulosa descrita anteriormente para selecionar o melhor conjunto de passageiros para o motorista D_i. A função *GetBenefit* simplesmente calcula o benefício total do conjunto de passageiros selecionados pelo *KnapsackGreedy*.

Algorithm 3 GreedyMaximizeBenefit

```
procedure GreedyMaximizeBenefit(G(V,E))
    Plan := ∅
    MaxBenefit := 0

    while G.E ≠ ∅ do
        BestDriver := null
        BestPlan := ∅
        MaxDriverBenefit := 0

        for all D_i ∈ G.Drivers do
            S := KnapsackGreedy(D_i, G.V)
            DriverBenefit := GetBenefit(S)
            if DriverBenefit > MaxDriverBenefit then
                MaxDriverBenefit = DriverBenefit
                BestDriver = D_i
                BestPlan = S
        Update Plan with the edges in BestPlan
        Remove inbound and outbound edges from best driver in graph G
        MaxBenefit = MaxBenefit + MaxDriverBenefit
    return MaxBenefit
```

Experimentos

Foram feitos experimentos com grafos completos variando entre 1 e 20 vértices em cada uma dos três paradigmas implementados. A figura **??** descreve o desempenho de cada um dos algoritmos propostos. No caso do algoritmo guloso também foi calculado quanto a solução gulosa se aproximou da solução ótima. O resultado está descrito na tabela **??**.

Conclusão

Este relatório descreveu a implementação do trabalho prático 2 da disciplina Projeto e Análise de Algoritmos. Entre as três abordagens propostas, a abordagem gulosa apresentou claramente o melhor desempenho chegando a uma aproximação média de 90% da resposta ótima em um tempo 100 vezes menor do que as abordagens exatas. A programação dinâmica não mostrou um ganho muito grande em relação à força bruta devido ao *overhead* na montagem da tabela de respostas. No entanto, talvez o resultado seja diferente para grafos significativamente maiores.

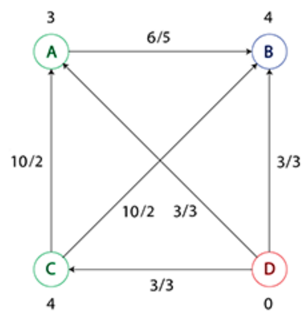

Figure 3: Número de vértices X Tempo de execução

Vértices	Algoritmo Guloso	Algoritmo Exato	Aproximação
1	65	233106	233106%
2	33	220774	220774%
3	22	220127	220127%
4	15	278959	278959%
5	13	211959	211959%
6	7	223152	223152%
7	4	247336	247336%
8	2	285716	285716%
9	2	235183	235183%
10	2	302967	302967%
11	2	302967	302967%
12	2	302967	302967%
13	2	302967	302967%
14	2	302967	302967%
15	2	302967	302967%
16	2	302967	302967%
17	2	302967	302967%
18	2	302967	302967%
19	2	302967	302967%
20	2	302967	302967%

Table 1: Aproximação pelo algoritmo guloso