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 16, 2017

1 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.

2 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.

3 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

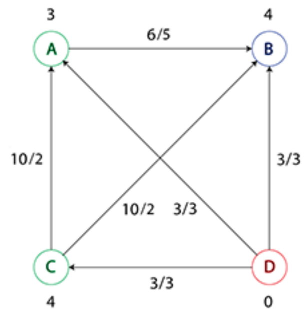

Figure 1: Exemplo do problema de compartilhamento de viagens modelado em forma de grafo

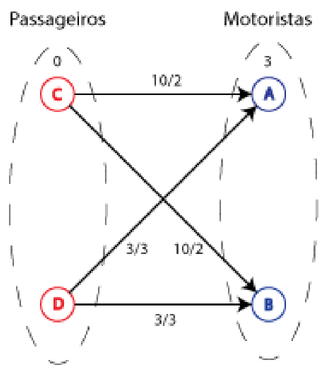

Figure 2: Grafo bipartido após definição dos passageiros e motoristas no grafo da figura 1

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 qual o benefício máximo para cada combinação de vértices no grafo G. Considere os motoristas $\{D_1, D_2, ..., D_n\}$. Agora, assuma uma combinação de passageiros válida em D_i que faz parte de um plano de benefício máximo $B_{max}(G, D_i)$ e remova de G todos os passageiros escolhidos por D_i e o próprio motorista D_i, gerando o grafo G'. Ora, $B_{max}(G, D_i) = B(D_i) + B_{max}(G', D_{i-1})$, onde $B(D_i)$ é a soma dos benefícios dos passageiros de D_i. Ou seja, o problema tem subestrutura ótima. A prova deste teorema é que, se $B(G, D_i)$ é ótimo mas a configuração que gera o benefício $B(G', D_{i-1})$ não é máxima, poderíamos substituí-la por uma configuraçãp que gerasse um benefício total maior, no entanto, isso contradiz a hipótese inicial que $B_{max}(G, D_i)$ era máximo. Então, $B_{max}(G, D_i)$ tem de ser máximo.

Todavia, nós não sabemos qual é a configuração de passageiros de benefício máximo para D_i. Portanto, devemos testar todas as possibilidades e escolher aquela que oferece o maior benefício. Finalmente, podemos construir a seguinte equação de recorrência:

$$B_{max}(G, D_i) = \begin{cases} 0, & se\ G = \emptyset. \\ \max\limits_{P_{comb} \subseteq D_i.P} B_{max}(G', D_{i-1}) + \sum\limits_{p \in P_{comb}} B(p), & caso\ contrário. \end{cases} \tag{1}$$

onde P_{comb} representa uma das combinações de passageiros do motorista D_i, e $D.P$ são todos os potencias passageiros do motorista D_i no grafo G. G' depende da escolha dos passageiros como explicado nos parágrafos anteriores e $B(P)$ é o benefício do passageiro P. A ordem de escolha dos motoristas D_i não é importante uma vez que todas as possíveis combinações de passageiros acabam sendo testadas de uma forma ou de outra.

A implementação ingênua desta equação de recorrência acaba calculando várias vezes o mesmo problema, pois, o poblema $B(G, D_i)$ pode ocorrer mais de uma vez na árvore de recursão. Portanto, podemos ganhar eficiência ao salvar os resultados dos cálculos em uma tabela.

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 $G.V$. 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 $v_k \in V_j \forall v_k \in G'.V$. O tamanho total da matriz B_{max} será $O(n2^n)$ sendo n o número de vértices do grafo G.

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 única informação nova é a função *GetCombIndex(G.V)* que retorna o índice j na tabela B_{max} para o subconjunto de vértices do grafo G. Para n igual ao número de vértices no grafo G, o loop principal é chamado no máximo n vezes. O loop secundário é chamado exatamente 2^n vezes e o terceiro loop é

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
                MaxBenefit := 0
                if i > 0 then
                    MaxBenefit := B_max[i − 1][j]    ▷ Benefit without adding any passengers.
                Remove D_i from G.V
                for all P_comb ⊆ D_i.P do
                    Benefit := 0
                    for all p ∈ P_comb do
                        Benefit := Benefit + B(p)
                        Remove p from G.V
                    CombIndex := GetCombIndex(G.V)
                    Benefit := Benefit + B_max[i − 1][CombIndex]
                    if Benefit > MaxBenefit then
                        MaxBenefit := Benefit
                    Add D_i back to G.V
                    for all p ∈ P_comb do
                        Benefit := Benefit + B(p)
                        Add p back to G.V
            else
                B_max[i][j] = 0
                if i > 0 then
                    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]
    return MaxBenefit
```

chamado no máximo 2^n vezes. Por último, o loop final é chamado exatamente 2^n vezes. Portanto, a complexidade do algoritmo é $n2^n2^n + 2^n = n2^{2n} + 2^n$ e o algoritmo é $O(n4^n)$. A complexidade de espaço do algoritmo é $O(n2^n)$ como exposto anteriormente, mas podemos reduzí-la para $O(2^n)$ se salvarmos apenas a linha anterior em cada iteração e, na iteração final, podemos simplesmente escolher a combinação que dê o benefício máximo sem salvar as respostas para cada combinação. Não é uma solução muito eficiente, mas já é bem melhor do que o algoritmo de força bruta. Devido ao fato do problema do compartilhamento de viagens ser NP-difícil, todas as soluções serão de ordem exponencial.

Para encontrar a configuração que gerou o benefício máximo, basta salvar o valor j da linha anterior sempre que acharmos uma configuração de benefício maior. Então, começando de trás para frente é só conferir quais vértices foram adicionados entre o melhor j da linha anterior e o j atual. Assim, saberemos que os elementos adicionados são os passageiros do motorista da linha atual. Para guardar os valores dos j anteriores precisaremos de uma tabela com o mesmo tamanho de B_{max}. O código referente a este trabalho está implementado conforme descrito neste parágrafo.

4 Algoritmo Guloso

O algoritmo guloso para o problema do compartilhamento de viagens o problema de escolher o subconjunto ótimo de passageiros para cada motorista a cada iteração como uma instância separada do *0-1 Knapsack Problem*. Basicamente, os itens da sacola (passageiros) serão ordenados por ordem não crescente de eficiência, sendo que a eficiência é definida por:

$$\frac{\textit{benefício}}{\textit{número de passageiros na viagem}}$$

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. Essa estratégia não garante uma solução ótima para o problema, mas é bastante rápida e tem complexidade $O(nlogn)$ devido à ordenação dos passageiros.

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 as arestas que saem dos passageiros e do motorista do grafo G, produzindo o grafo G'. Repetimos o processo até que não haja mais motoristas no grafo e, ao final do processo, obtemos o benefício total. Da mesma forma que o *0-1 Knapsack Problem*, esta estratégia não produz necessariamente um resultado ótimo. No entanto, a complexidade para a escolha do motorista é n^2logn e este processo é repetido no máximo n vezes, portanto, a complexidade do algoritmo é $O(n^3logn)$, que é polinomial e, portanto, consideravelmente melhor do que as alternativas exatas.

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 edges from best driver and all passengers in graph G
        MaxBenefit = MaxBenefit + MaxDriverBenefit
    return MaxBenefit
```

A abordagem gulosa para o problema da maximização de benefício no grafo de compartilhamento de viagens está descrita no algoritmo 3. 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*.

5 Experimentos

Foram feitos experimentos com grafos completos variando entre 1 e 20 vértices em cada um dos três paradigmas implementados. Todos os grafos foram gerados aleatoriamente, sendo que todos os vértices podem sempre ser passageiros ou motoristas. A figura 3 descreve o desempenho dos algoritmos propostos nos casos de teste. Os algoritmos de força bruta e programação dinâmica sempre chegam na solução exata, sendo que o paradigma de força bruta começou a se tornar impraticável a partir de 10 vértices e 90 arestas e o paradigma de programação dinâmica começou a se mostrar impraticável a partir de 16 vértices e 240 arestas. O algoritmo guloso tem complexidade polinomial e é viável para exemplos muito maiores do que os que foram aqui apresentados, no entanto, ele nem sempre chega na solução exata. No caso do algoritmo guloso também foi calculado quanto a solução gulosa se aproximou da solução ótima. Nos testes com até 15 vértices, o algoritmo guloso conseguiu uma aproximação média de

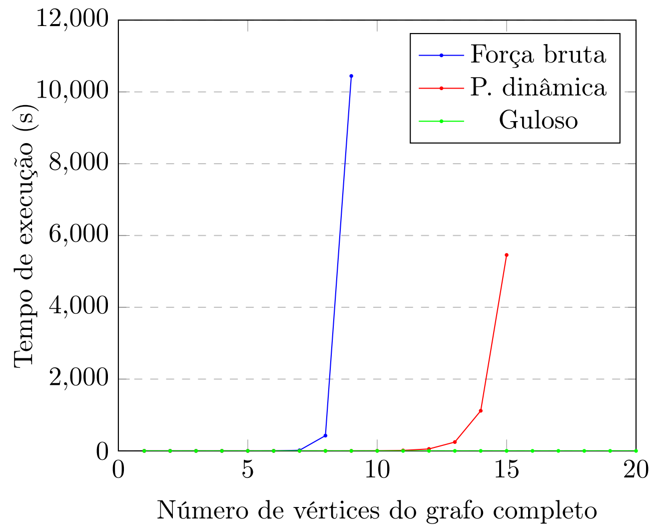

Figure 3: Desempenho das abordagens força bruta, programação dinâmica e gulosa

V	A	Exato	Guloso	Aproximação	Exato (ms)	Guloso (ms)
1	1	0.0	0.0	100%	0	0
2	2	70.32	70.32	100%	0.18	0.04
3	6	37.08	37.08	100%	0.47	0.05
4	12	0.0	0.0	100%	0.68	0.11
5	20	107.32	106.94	99.64%	4.5	0.08
6	30	269.12	269.12	100%	15	0.14
7	42	142.29	142.29	100%	37	0.12
8	56	209.23	209.23	100%	171	0.25
9	72	234.84	219.12	93.31%	677	0.32
10	90	218.22	218.22	100%	2758	0.53
11	110	188.83	167.51	88.71%	13202	0.51
12	132	461.61	418.21	90.59%	54524	0.42
13	156	384.5	288.04	74.91%	246305	0.71
14	182	448.55	408.28	91.02%	1117631	0.64
15	210	611.33	464.24	75.94%	5457820	0.63
16	240	-	497.9	-	-	0.69
17	272	-	384.05	-	-	0.67
18	306	-	376.7	-	-	0.95
19	342	-	729.13	-	-	0.89
20	380	-	492.84	-	-	1.29

Table 1: Aproximação pelo algoritmo guloso

94.27% da solução exata em um tempo inferior a 1 milisegundo. O resultado está descrito na tabela 1.

6 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 94.27% da resposta ótima em um tempo muitas ordens de magnitude menor do que as abordagens exatas. A programação dinâmica mostrou um ganho muito grande de desempenho em relação ao algoritmo de força bruta, no entanto, a abordagem gasta uma quantidade muito maior de memória.