Material didático sobre algoritmos gulosos

Trabalho de conclusão de curso
Universidade de São Paulo
Bacharelado em Ciência da Computação

Victor de Oliveira Colombo
Orientador: Carlos Eduardo Ferreira

São Paulo, 2018

Resumo

Algoritmos gulosos são uma classe de algoritmos muito importantes para a resolução de problemas de otimização. Nesses algoritmos, a solução ótima global é atingida através de escolhas localmente ótimas, também conhecidas como escolhas gulosas.

Embora este tópico esteja presente na ementa de qualquer curso introdutório de algoritmos, muitos alunos têm dificuldades para discernir quando utilizá-lo em detrimento das demais técnicas. Nestes cursos, normalmente são apresentados apenas problemas clássicos, como *Activity selection*, e com livro texto em língua estrangeira.

Foi produzido um material didático em português focado no ensino de técnicas por meio de resolução de problemas desafiantes e não usuais, como vistos em competições como Maratona de Programação e ICPC, a fim de desenvolver no leitor a capacidade de identificar se um problema pode ser resolvida usando uma estratégia gulosa, propor tal solução e implementá-la, transformando este processo em algo sistemático.

Palavras-chave: algoritmos gulosos, programação competitiva, material didático, otimização.

Agradecimento

Primeiramente, agradeço ao professor Carlos Eduardo Ferreira por ter sido meu orientador neste trabalho e meu mentor durante todo o curso.

Agradeço aos meus amigos Giovanna Kobus e Lucas Daher por fornecerem constantes comentários construtivos para a melhoria deste trabalho.

Agradeço também a todos os membros do grupo MaratonIME[1] que me motivaram a continuar resolvendo problemas e aprendendo novos algoritmos e estruturas de dados.

Por fim, agradeço profundamente minha família que sempre esteve ao meu lado, incentivando e possibilitando a realização de todos meus sonhos pessoais, acadêmicos e profissionais.

[1] https://www.ime.usp.br/~maratona/

Sumário

Capítulo 1

Introdução

Algoritmos gulosos são uma popular escolha para resolução de problemas de otimização, com um dos primeiros registros de sua utilização datando de 1952, com a Codificação de Huffman [5].

Abordagens gulosas nem sempre produzem a solução ótima, mas quando produzem, são mais eficientes quando comparadas com, por exemplo, Programação Dinâmica ou força bruta.

Eles são muito difíceis de se definir precisamente. Há pesquisa para formalizar a classe de problemas considerados gulosos, como os trabalhos de Edmonds [4] e Lawler [7] na teoria dos matroides, e o trabalho de Borodin, Nielsen e Rackoff [1] na definição de "guloso".

Em seu livro didático, Cormen [2] define algoritmos gulosos como algoritmos que apresentam *propriedade de escolha gulosa*: conseguimos chegar na solução ótima global através de escolhas localmente ótimas (escolhas gulosas). Isto é, em contraste com a programação dinâmica, que cada escolha depende dos subproblemas, num algoritmo guloso fazemos a escolha que parece melhor para a situação atual, sem considerar os subproblemas, e, mais importante, uma vez que uma decisão é tomada, não a mudamos mais.

1.1 Objetivos

Neste trabalho apresentamos algoritmos gulosos de uma maneira pragmática, a partir de problemas de competições de programação, como a Maratona de Programação[1], destoando dos problemas clássicos que são frequentemente abordados nos livros-texto.

Além de apresentar os conceitos paralelamente às soluções para os problemas, focamos também em desenvolver o raciocínio e a intuição por trás das técnicas gulosas, a fim de criar uma ferramenta sistemática para resolvê-los.

1.2 Estrutura do texto

No capítulo 2 apresentamos o argumento de troca, uma ferramento versátil para demonstração de corretude de algoritmos gulosos que será utilizada nos demais capítulos.

Nos capítulos 3 e 4 investigamos como podemos reduzir problemas a ponto de aceitarem soluções gulosas. Discutimos também como a utilização de estruturas de dados auxiliares aliado ao algoritmo

[1] http://maratona.ime.usp.br/

guloso pode diminuir sua complexidade computacional.

No capítulo 5 mostramos como algoritmos gulosos e programação dinâmica, duas técnicas distintas para resolução de problemas de otimização, podem ser utilizadas em conjunto.

No capítulo 6 exploramos como restrições na entrada de um problema podem mudar completamente a natureza de sua solução, podendo admitir soluções gulosas que não seriam possíveis sem as restrições.

Além do problema analisado, cada capítulo propõe exercícios teóricos de aprofundamento no conteúdo e exercícios práticos no estilo programação competitiva.

1.3 Programação competitiva

Programação competitiva é um esporte da mente onde os participantes devem resolver tarefas, normalmente relacionadas à lógica, algoritmos e estruturas de dados, dentro de um período de limitado tempo, seguindo restrições de processamento e espaço.

Uma das competições mais antigas é a *International Collegiate Programming Contest (ICPC)*, que reúne estudantes de milhares de universidades de centenas de países ao redor do mundo, competindo em trios num mesmo computador durante cinco horas. Existem diversas competições que seguem o modelo da *ICPC*, como *Northwestern Europe Regional Contest (NWERC)*, *Benelux Algorithm Programming Contest (BAPC)*. Problemas destas competições são disponibilizados em arquivos de problemas chamados "juízes", como *LiveArchive*, *Kattis* e *SPOJ*.

Outro modelo popular de competição são os *contests online*, que normalmente são individuais e tem periodicamente semanal ou bissemanal, hospedadas e promovidas por sites como o *CodeForces* e *CodeChef*.

Os problemas práticos propostos neste trabalho estão disponíveis nesses "juízes". Além de resolver teoricamente, é possível implementar numa linguagem de programação e submeter para uma correção automática, que compara a saída do programa submetido com as saídas esperadas.

Os vereditos mais comuns são *Accepted* ou *AC*, quando o programa está correto, *Wrong Answer* ou *WA*, quando o programa produz resposta errada para algum caso, e *Time Limit Exceeded* ou *TLE*, quando a solução entra em *loop* infinito ou não possui complexidade computacional adequada para as restrições.

Capítulo 2

Argumento de troca

É muito comum que problemas que aceitam soluções gulosas possuam diversas outras soluções ótimas. Assim, demonstrações por contradição que assumem a existência de uma solução ótima que é diferente da solução gulosa são insuficientes, já que solução escolhida também pode ser ótima, mas distinta da gulosa.

Visando superar esta dificuldade, podemos aplicar uma técnica conhecida como "argumento de troca". Esta consiste em escolher uma solução ótima conveniente. Tomamos uma solução ótima "mais parecida" com a solução gulosa possível. Isto é, escolhemos uma solução ótima com maior prefixo de escolhas em comum com a solução gulosa. Desta forma, se a solução gulosa for ótima, elas coincidirão.

Buscamos a contradição assumindo que as escolhas diferem em algum momento. Daí buscamos "trocar" a decisão da solução ótima pela solução gulosa e mostramos que tal troca não altera o resultado, contrariando a hipótese do maior prefixo de escolhas em comum.

A seguir aplicaremos tal técnica para demonstrar a corretude da solução para um problema.

2.1 Salto do sapo

Existem pedras n pedras numa reta numérica, em posições distintas $v_1, v_2, ..., v_{n-1}, v_n$. Dizemos que o sapo pode saltar de uma pedra v_i para outra pedra v_j desde que a distância entre elas seja menor ou igual a Δ. Um sapo está inicialmente na pedra v_1. Qual é o menor número de saltos que ele precisa dar para chegar na pedra v_n?

Ou seja, é dado um vetor de n números distintos ordenados $v = \{v_1, v_2, ..., v_n\}$ e um número Δ. Uma sequência $u = \{u_1, u_2, ..., u_k\}$ é **solução** se:

- $u_1 = v_1$
- $u_k = v_n$
- $u_i = v_j$ para todo $i \in [1, k]$ e algum $j \in [1, n]$
- $|u_i - u_{i+1}| \leq \Delta$ para $i \in [1, k-1]$

Vamos supor que sempre existe pelo menos uma solução.

Desejamos encontrar uma sequência u que satisfaça as propriedades acima e que o tamanho k de u seja mínimo.

Chamamos este u de solução ótima para o problema.

Exemplos

i) A entrada $n = 4$, $v = \{1, 2, 3, 4\}$, $\Delta = 1$ existem diversas soluções possíveis, entre elas $\{1, 2, 3, 4\}$, $\{1, 2, 1, 2, 3, 4\}$ e $\{1, 2, 1, 2, 3, 2, 3, 4\}$. A sequência de menor $k = 4$ é $u = \{1, 2, 3, 4\}$.

ii) A entrada $n = 6$, $v = \{1, 2, 3, 5, 6, 7\}$, $\Delta = 2$ tem como solução ótima $u = \{1, 3, 5, 7\}$.

iii) A entrada $n = 3$, $v = \{1, 3, 4\}$, $\Delta = 1$ não admite solução, já que a partir de 1 não é possível que 3 ou 4 sejam o próximo elemento da sequência. Como dissemos, desconsideraremos estes casos neste problema.

2.2 Desenvolvimento

Primeiramente devemos notar que nunca vale a pena *"voltar"*, isto é, escolher um número menor que o escolhido anteriormente, pois isto aumentaria desnecessariamente a sequência, já que poderíamos descartar a escolha anterior e escolher apenas o menor número.

Além disso, devemos fazer a observação que, para todo $y \in v$, nenhum número maior que y pode ser sucessor de algum número menor que y sem que pudesse ser sucessor do próprio y. Ou seja, intuitivamente não existe *"vantagem"* em escolher um número menor que o maior possível.

Partindo das observações anteriores, uma solução intuitiva é *"ir mais para direita possível"*: Partindo de $u_1 = v_1$, escolha o próximo u_{i+1} tal que a diferença ao elemento anterior, $u_{i+1} - u_i$, seja máxima e menor ou igual a Δ, repetindo até escolher v_n.

Neste caso, felizmente, a primeira ideia que consideramos é de fato a correta. Agora o trabalho especificar o algoritmo e demonstrar sua corretude.

2.2.1 Algoritmo

A ideia descrita acima é simples de se implementar, basta manter o último v_i selecionado para u e atualizá-lo assim que um v_j exceder a distância Δ.

Algoritmo 2.1 Solução gulosa para o Problema 2

```
1: função RESOLVE(v, n, Δ)
2:     u_1 ← v_1
3:     j ← 2
4:     para i de 2 até n faça
5:         se v_i − u_j > Δ então
6:             u_j ← v_{i−1}
7:             j ← j + 1
8:     u_j ← v_n
9:     devolve u
```

É fácil ver que o pseudocódigo acima tem complexidade de tempo $O(n)$ e de memória $O(1)$.

2.2.2 Demonstrações

Proposição 2.2. *Se u é ótima, u é estritamente crescente.*

Demonstração. Se $k = 1$, está provado.

Se $k = 2$, u é estritamente crescente e ótima por definição, já que $u_1 = v_1 < v_n = u_k$

Suponha que $k \geq 3$ e que u é ótima mas não é crescente. Seja i o primeiro índice tal que $u_i > u_{i+1}$. Da definição, segue que:

$$0 \leq u_i - u_{i+1} \leq \Delta \tag{2.3}$$

Além disso, temos que $\{u_1, u_2, ..., u_i\}$ é crescente. Ou seja, vale que $u_{i-1} < u_i$. Como u é solução, segue que:

$$0 \leq u_i - u_{i-1} \leq \Delta \tag{2.4}$$

Multiplicando a equação 2.3 por -1 e somando com 2.4 temos:

$$\begin{aligned} -\Delta \leq -u_i + u_{i+1} \leq 0 \\ 0 \leq u_i - u_{i-1} \leq \Delta \\ -\Delta \leq u_{i+1} - u_{i-1} \leq \Delta \\ |u_{i+1} - u_{i-1}| \leq \Delta \end{aligned} \tag{2.5}$$

Da equação 2.5 segue que $\bar{u} = \{u_1, u_2, ..., u_{i-1}, u_{i+1}, ..., u_k\}$ é solução de tamanho $k - 1$. Contradição, já que por hipótese k é mínimo.

□

Teorema 2.6. *O algoritmo 2.1 produz uma solução ótima.*

Demonstração. Suponha que o algoritmo proposto produz uma solução u de tamanho k. Sabemos pela proposição anterior que u é estritamente crescente.

Seja u^* de tamanho k^* uma solução ótima com maior prefixo comum com u. Ou seja, se $u_i^* = u_i$ para $i \in [1, l]$, u^* é tal que l é máximo. Como u^* é ótima, temos que $k^* \leq k$. Sabemos pela proposição 2.2 que u^* é estritamente crescente.

Provaremos por contradição que $k^* = k$. Suponha que $k^* < k$.

Se $l = k^*$, temos uma contradição, pois como $u_{k^*}^* = u_l^* = v_n$ e $u_l^* = u_l$, então $u_l = u_k$, contrariando a condição de parada do algoritmo.

Se $l < k^*$, temos que $u_l^* = u_l$, $u_{l+1}^* \neq u_{l+1}$. Pelo critério de escolha do algoritmo, temos que $u_{l+1} > u_{l+1}^*$.

Caso 1: $l + 1 = k^*$. Como $u_{l+1}^* = v_n$ e v_n é o maior elemento de v, não existe escolha tal que $u_{l+1} > v_n$. Contradição.

Caso 2: $l + 1 < k^*$. Como as sequências u e u^* são válidas, valem as seguintes identidades:

$$u_{l+2}^* - u_{l+1}^* \leq \Delta$$

$$u^*_{l+1} - u^*_l \leq \Delta$$

$$u_{l+1} - u_l \leq \Delta$$

Também, como tanto u quanto u^* são estritamente crescentes, $u_{l+1} > u^*_{l+1}$, concluímos que $|u^*_{l+2} - u_{l+1}| \leq \Delta$.

Desta forma, podemos **trocar** u^*_{l+1} por u_{l+1}, criando a sequência $\bar{u} = \{u_1, u_2, ..., u_l, u_{l+1}, u^*_{l+2}, ..., u^*_{l*}\}$. Como $\bar{u}$ tem o mesmo tamanho que u^* e possui prefixo comum com u de tamanho $l+1$, temos uma contradição na hipótese que l é máximo.

□

2.3 Exercícios

Teóricos

1. Modifique o algoritmo para que ele trate quando a instância não possui resposta. Prove que sua modificação está correta, ou seja, produz uma solução se e somente se há uma solução.

2. Suponha que v possui inteiros repetidos e não necessariamente em ordem. É possível generalizar a solução? Demonstre.

Problemas

3. Partition - Educational Codeforces Round 39 (Rated for Div. 2)

4. String Transformation - Educational Codeforces Round 39 (Rated for Div. 2)

5. The Glittering Caves of Aglarond - Asia Amritapuri Regional Contest 2012

6. The Modcrab - Educational Codeforces Round 34 (Rated for Div. 2)

Capítulo 3

Ordenação gulosa

No capítulo anterior apresentamos um problema que utilizava um critério guloso para escolha direta da resposta. Nesse capítulo abordaremos um problema que envolve não um, mas dois critérios gulosos em etapas distintos do algoritmo para sua resolução, sendo que um deles é uma ordenação gulosa para a tomada de decisão do algoritmo.

3.1 Escalonamento minimizando a multa total

São dadas tarefas $1, 2, ..., n$ que demoram uma unidade de tempo cada para serem completadas. A tarefa i tem um prazo $p_i \in \{1, 2, ..., n\}$ e um multa $m_i \geq 0$ caso seja realizada após o prazo. Desejamos escalonar cada tarefa a exatamente uma unidade de tempo em $\{1, 2, ..., n\}$ tal que a multa total seja mínima.

Definimos que, se uma tarefa i é escalonada para um tempo anterior ou igual ao seu prazo p_i, não pagamos nada, caso contrário pagamos a multa m_i. A multa total é definida como a soma das multas das tarefas que não foram escalonadas até seus prazos.

Exemplos

i) São dadas três tarefas: tarefa 1 tem prazo $p_1 = 2$ e multa $m_1 = 1$, tarefa 2 tem prazo $p_2 = 2$ e multa $m_2 = 2$, tarefa 3 tem prazo $p_3 = 2$ e multa $m_3 = 3$.
Como todos as tarefas têm o mesmo prazo 2, uma delas com certeza pagará multa. Para minimizar a multa total, basta fazer com que a tarefa de menor multa ser escalonada depois do prazo.
Assim, temos a multa mínima igual a $m_1 = 1$ no seguinte escalonamento: tarefa 3 em $t = 1$, tarefa 2 em $t = 2$ e tarefa 1 em $t = 3$.
Note que o seguinte escalonamento também apresenta a mesma multa que o anterior: tarefa 2 em $t = 1$, tarefa 3 em $t = 2$, tarefa 1 em $t = 3$.

ii) São dadas três tarefas: tarefa 1 tem prazo $p_1 = 1$ e multa $m_1 = 3$, tarefa 2 tem prazo $p_2 = 2$ e multa $m_2 = 2$, tarefa 3 tem prazo $p_3 = 3$ e multa $m_3 = 1$.
É possível ter multa total 0 com o seguinte escalonamento: tarefa 1 em $t = 1$, tarefa 2 em $t = 2$ e tarefa 3 em $t = 3$.

3.2 Desenvolvimento

Vamos supor que um conjunto de tarefas já foi escalonado e que desejamos escalonar uma tarefa i buscando não pagar sua multa, se possível.

Uma primeira observação a ser feita é que se não existe tempo $t \leq p_i$ "livre", isto é, que já não tenha sido escalonado para outra tarefa, estamos fadados a pagar a multa m_i. Assim, queremos colocar a tarefa i num tempo que menos "atrapalhe" o escalonamento de outras tarefas. Para isso, podemos seguir uma estratégia de quanto mais tarde, melhor, escolhendo o maior tempo ainda disponível.

Outra observação é que se existem múltiplos instantes de tempo $t \leq p_i$ disponíveis, mesmo que não paguemos a multa m_i, ainda nos resta encontrar qual o melhor instante de tempo para acomodá-la. De maneira análoga à observação anterior, gostaríamos de escalonar a tarefa i para mais tarde possível que não pague multa, ou seja, o maior $t' \leq p_i$ disponível.

Destas observações segue o seguinte algoritmo:

Evitando multa e quanto mais tarde, melhor

Partindo da tarefa 1, verifique se é possível não pagar sua multa. Em caso afirmativo, escalone para o maior tempo $t_1 \leq p_1$. Caso contrário, escalone para o tempo mais tarde ainda livre. Repita para 2, 3, ..., até n. Esta ideia pode ser expressa pelo seguinte pseudocodigo:

Algoritmo 3.1 Solução gulosa errada para o Problema 3

```
1:  função RESOLVE(v, n)
2:      escalonamento ← {0}^n
3:      para i de 1 até n faça
4:          para j de n até 1 faça
5:              se escalonamento[j] = 0 então
6:                  t ← j
7:                  break
8:          para j de v[i].p até 1 faça
9:              se escalonamento[j] = 0 então
10:                 t ← j
11:                 break
12:         escalonamento[t] ← v[i].indice
13:     devolve escalonamento
```

Encontramos assim um critério de escolha de tempos que aplica uma ideia gulosa de evitar pagar multa quando possível. Entretanto, a estratégia proposta não leva a uma solução ótima.

Um simples contra exemplo para tal algoritmo é o caso $p_1 = 1$, $m_1 = 1$, $p_2 = 1$, $m_2 = \infty$. A tarefa 1 pode ser escalonada no tempo $t = 1 \leq p_1$ evitando pagar a multa $m_1 = 1$ mas fazendo com que a tarefa 2 fosse escalonada fora do seu prazo, acarretando numa multa total muito maior. Neste exemplo, gostaríamos pagar a multa da tarefa 1 pois isso possibilita evitar de pagar a multa da tarefa 2, que é muito mais vantajosa globalmente.

Percebemos que sem alguma observação adicional, a escolha do mínimo local, deixando de pagar a multa da tarefa atual, não acarreta necessariamente o mínimo global, ou seja, não minimiza a soma total das multas pagas. Como abordado na introdução, este é um princípio fundamental para aplicação de técnicas gulosas.

Utilizando o contra exemplo acima, desenvolvemos a intuição de que vale a pena "acomodar" primeiro tarefas com maiores multas.

Evitando a maior multa e quanto mais tarde, melhor

Processando as tarefas da maior multa para menor, verifique para cada uma se é possível não pagar sua multa. Em caso afirmativo, escalone tal tarefa para o maior tempo anterior ou igual ao seu prazo. Caso contrário, escalone-a para o tempo mais tarde ainda livre.

Esta abordagem está correta. Implementaremos e provaremos sua corretude a seguir.

3.2.1 Algoritmo

Para implementação deste problema, podemos montar um vetor *escalonamento* tal que $escalonamento[j] = i$ indica que a tarefa i foi atribuída ao tempo j. Se o tempo j não foi escalonado para nenhuma tarefa numa dada iteração, assumimos que $escalonamento[j] = 0$.

Para cada tarefa i, montaremos uma tripla $v_i = (m = m_i, p = p_i, indice = i)$. Ordenaremos tais triplas em ordem decrescente de m, com empates resolvidos ao acaso.

Agora, processando a partir da tripla com maior multa até a menor, para cada tarefa i, procuramos se existe $escalonamento[j] = 0$ tal que $j \leq m_i$. Se existe, tomamos o maior j que satisfaz tal condição e atualizamos $escalonamento[j] = i$. Caso contrário, tomamos o j o maior tempo tal que $escalonamento[j] = 0$ e atualizamos $escalonamento[j] = i$. Cada tarefa requer $O(n)$ para encontrar tal j, resultando no complexidade de tempo $O(n^2)$.

Algoritmo 3.2 Solução gulosa ingênua para o Problema 3

```
1:  função RESOLVE(v, n)
2:      Ordene(v)                          ▷ ordena tarefas em ordem decrescente de multa
3:      escalonamento ← {0}^n
4:      para i de 1 até n faça
5:          para j de n até 1 faça
6:              se escalonamento[j] = 0 então
7:                  t ← j
8:                  break
9:          para j de v[i].p até 1 faça
10:             se escalonamento[j] = 0 então
11:                 t ← j
12:                 break
13:         escalonamento[t] ← v[i].indice
14:     devolve escalonamento
```

3.2.2 Demonstrações

Assumiremos sem perda de generalidade que $m_1 \geq m_2 \geq ... \geq m_n$.

Um escalonamento e é uma permutação de $\{1, 2, ..., n\}$ tal que $e_i = j$ significa que a tarefa i foi escalonada para o tempo j. Por definição, $e_i \neq e_j$ para todo $i \neq j$ e $e_i \in \{1, 2, ..., n\}$. Definimos como $M(e)$ a multa total deste escalonamento.

Como vimos no capítulo anterior, podemos demonstrar que nosso algoritmo produz um escalonamento ótimo através de um argumento de troca.

Seja e o escalonamento produzido pelo algoritmo descrito e e^* o escalonamento ótimo com maior prefixo comum com e, ou seja, e^* é tal que $M(e^*)$ é mínimo e $e_i = e_i^*$ para todo $i \in [1, k-1]$ com k máximo.

Se $e = e^*$, está provado. Caso contrário, seja k o primeiro índice tal que $e_k \neq e_k^*$.

Aqui é necessário fazer uma análise caso a caso.

Caso $e_k < e_k^*$ e tarefa k não paga multa em e

Por definição, e_k é o último tempo possível em que a k-ésima tarefa pode ser escalonada sem multa. Assim, a k-ésima tarefa paga multa em e^* mas não em e.

Seja x a tarefa tal que $e_x^* = e_k$. Montando um escalonamento $\widetilde{e}$ trocando os tempos das tarefas x e k em e^*, temos que a tarefa k agora não cobra multa, já que $\widetilde{e}_k = e_k$ e a tarefa x pode cobrar ou não, já que $\widetilde{e}_x > e_x^*$.

Ora mas, por hipótese, vale que $m_x \leq m_k$. Da identidade:

$$M(e^*) - m_k \leq M(\widetilde{e}) \leq M(e^*) - m_k + m_x \leq M(e^*)$$

E do fato que $\widetilde{e}$ tem prefixo comum com e de tamanho maior que e^*, há uma contradição.

Caso $e_k < e_k^*$ e tarefa k paga multa em e

Se a tarefa k paga multa em e e $e_k < e_k^*$, ela paga multa em e^*, o que é uma contradição na escolha do algoritmo, já que e_k é o maior instante de tempo livre.

Caso $e_k > e_k^*$ e tarefa k não paga multa em e

Se a tarefa k não paga multa em e e $e_k > e_k^*$, ela não paga multa em e^*. Seja x a tarefa tal que $e_x^* = e_k$. Montando um escalonamento $\widetilde{e}$ trocando os tempos das tarefas x e k em e^*, temos que a tarefa k continua não cobrando multa, já que $\widetilde{e}_k = e_k$ e se a tarefa x não cobrava multa em e^*, não cobra multa em $\widetilde{e}$, já que $\widetilde{e}_x = e_k^* \leq e_x^*$.

Analogamente ao caso anterior, temos que $M(\widetilde{e}) \leq M(e^*)$ e $\widetilde{e}$ tem prefixo comum com e de tamanho maior que e^*, seguindo assim uma contradição.

Caso $e_k > e_k^*$ e tarefa k paga multa em e

Seja x a tarefa tal que $e_x^* = e_k$. Montando um escalonamento $\widetilde{e}$ trocando os tempos das tarefas x e k em e^*, temos que a tarefa k cobra multa, já que $\widetilde{e}_k = e_k$ e se a tarefa x não cobrava multa em e^*, não cobra multa em $\widetilde{e}$, já que $\widetilde{e}_x = e_k^* \leq e_x^*$.

Da mesma forma que o caso anterior, temos que $M(\widetilde{e}) \leq M(e^*)$ e $\widetilde{e}$ tem prefixo comum com e de tamanho maior que e^*, seguindo assim uma contradição.

Como em todos os quatro casos chegamos numa contradição, o algoritmo produz a resposta ótima.

3.2.3 Otimizações

O gargalo do algoritmo 3.2 é a escolha do tempo, já que a ordenação das tarefas é $O(n \lg n)$. Podemos otimizar a escolha do tempo através do uso de uma estrutura de dados auxiliar que consegue realizar rapidamente as operações de inserção, remoção, retornar o maior elemento no conjunto e retornar o maior elemento menor que um dado número. Um exemplo de tal estrutura é uma Árvore de Busca Binária balanceada [8], que realiza todas as operações descritas acima em $O(\lg n)$, resultando numa complexidade total de $O(n \lg n)$.

Algoritmo 3.3 Solução gulosa para o Problema 3

```
1:  função RESOLVE(v, n)
2:      Ordene(v)                          ▷ ordena tarefas em ordem decrescente de multa
3:      escalonamento ← {0}^n
4:      abb ← ArvoreBuscaBinaria({1, 2, ..., n})
5:      para i de 1 até n faça
6:          t ← abb.MaiorMenor(v[i].p)
7:          se t < v[i].p então
8:              escalonamento[t] ← v[i].indice
9:          senão
10:             t ← abb.Maior()
11:             escalonamento[t] ← v[i].indice
12:         abb.Deletar(t)
13:     devolve escalonamento
```

3.3 Considerações finais

Percebemos que este problema é mais sofisticado comparado ao que foi apresentado no capítulo anterior. Diferentemente dos saltos, que apresentavam uma estrutura de escolhas independentes, agora precisamos lidar com o fato de que a escolha de um escalonamento para um conjunto de tarefas influencia a escolha para próxima tarefa.

Para remover esta dependência entre tarefas, precisávamos não só escolher um escalonamento para cada tarefa, mas também em que ordem essas escolhas eram feitas.

Além disso, foi necessário introduzir o uso de estruturas de dados mais sofisticadas na implementação para reduzir a complexidade computacional.

3.4 Exercícios

Teóricos

1. Mostre que a ordenação das arestas por menor peso no algoritmo de Kruskal produz de fato uma floresta geradora mínima de um grafo.

Problemas

2. Dentista - Olimpíada Brasileira de Informática 2010 (Fase 2, Nível Junior)
3. Getting an A - Codeforces Round #491 (Div. 2)
4. Reduzindo detalhes em um mapa - Olimpíada Brasileira de Informática 2011 (Fase 2, Nível 2)
5. O Código de Cormen - Maratona Mineira 2013
6. Balloons - Southeast USA Regional Intercollegiate Programing Contest 2010

Capítulo 4

Subproblemas gulosos

Nos capítulos anteriores, foram abordados problemas cuja resolução era a aplicação direta de um critério guloso, seja de escolha ou ordenação. Neste capítulo, o problema apresentado não terá solução gulosa mas, através de aplicação de outras técnicas, podemos reduzir o problema a um subproblema que aceita solução gulosa.

4.1 Operações em vetor

Um floricultor tem n flores dispostas sequencialmente numa linha reta, sendo que cada flor tem uma altura inicial. Ele tem X dias até a entrega de sua próxima encomenda. A cada dia é escolhido um intervalo de flores consecutivas para serem regadas. Como sua mangueira tem potência limitada, então ela só tem um alcance de $D+1$ flores. Isso é, se posicionarmos a mangueira na flor 3, regamos todas as flores no intervalo $[3, D+3]$. As flores, quando regadas por um dia, aumentam de tamanho em uma unidade.

Seu objetivo é utilizar **todas** as flores para montar o buquê mais bonito possível. A beleza de um buquê é definida como o tamanho da menor flor, ou seja, quando maior for a menor flor, mais bonito será o buquê.

Ou seja, dado um vetor de inteiros $h = \{h_1, h_2, ..., h_n\}$ e dois inteiros $X \geq 0$ e $0 \leq D \leq n-1$. Você pode fazer X operações em h. Cada operação é definida como incrementar em 1 todos os elementos de um intervalo $[i, min(i+D, n)]$, isto é, $h_i \leftarrow h_i+1$, $h_{i+1} \leftarrow h_{i+1}+1$, ..., $h_{min(i+D,n)} \leftarrow h_{min(i+D,n)}+1$.

Queremos encontrar o maior H tal que existe uma escolha de X operações sobre h onde vale que $h_i \geq H$ para qualquer $i \in [1, n]$.

Exemplos

i) Para $h = \{1, 2, 1\}$, $X = 2$ e $D = 2$, podemos aplicar uma operação no par h_1, h_2 e outra operação no par h_2, h_3, resultando no vetor $\{2, 4, 2\}$. Assim, podemos tomar $H = 2$ pois é maior ou igual a todos os elementos de $\{2, 4, 2\}$. Neste caso, $H = 2$ é máximo, pois com 2 operações de incremento em intervalos de tamanho 2 não é possível obter valor maior.

ii) Para $h = \{1, 2, 1\}$, $X = 5$ e $D = 1$, podemos aplicar duas operações em h_1 e três operações em h_3, resultando no vetor $\{3, 2, 4\}$. Assim, podemos tomar $H = 2$ pois é maior ou igual a todos

os elementos de $\{3, 2, 4\}$. Por outro lado, poderíamos ter aplicado duas operações em h_1, duas operações em h_3 e uma operação em h_2, resultando no vetor $\{3, 3, 3\}$. Assim, teríamos $H = 3$ que é o valor máximo neste caso.

4.2 Desenvolvimento

Imediatamente conseguimos perceber que este problema tem muitas "partes soltas", dificultando a aplicação direta de um algoritmo guloso. Precisamos encontrar o H máximo, que depende do h final, que por sua vez depende das operações escolhidas.

Uma abordagem útil quando estamos estagnados é modificar o problema para outro que parece mais simples e, depois de resolvê-lo, tentar generalizar ou modificar a solução para resolver o problema original. Neste caso, vamos fixar a variável resposta H.

Suponha que H é fixo, ou seja, gostaríamos de saber se é possível transformar h num vetor tal que H é o seu mínimo após realizar até X operações. Note que transformamos um problema de otimização em um problema de decisão.

Algoritmo guloso para H fixo

Quando fixamos o valor de H, podemos desenvolver um algoritmo guloso.

Se $h_1 < H$, precisamos aplicar pelo menos $H - h_1$ operações para que h_1 satisfaça a restrição. Note que não há escolha na realização de tais operações, já que o único intervalo de tamanho D que contém 1 é $[1, min(1 + D)]$. Se $h_1 \geq H$, não há nada a ser feito.

Aplicamos a mesma lógica para h_2. Note que agora temos duas opções de intervalos que contém 2, se $D > 0$: $[1, min(1+D, n)]$ e $[2, min(2+D, n)]$. Podemos, porém, descartar o intervalo $[1, min(1+D, n)]$ pois, por construção, h_1 já está correto. Assim, como $[2, min(2 + D, n)]$ pelo menos o número de elementos menores que H que $[1, min(1 + D, n)]$, podemos sempre escolher o intervalo com início no elemento.

É razoável pensar que este é o método que utiliza menos operações para transformar o vetor h tal que seu mínimo é H. Assim se o número de operações aplicadas forem menores ou iguais X, H é uma resposta viável.

Maximizando H

Agora que já sabemos como resolver o problema de decisão para H fixo, como achamos o maior H possível? Uma ideia ingênua seria iterar sobre todos os valores possíveis de H.

É fácil encontrar o intervalo de valores que H pode assumir. Primeiramente vale lembrar que, por definição, de H é o valor mínimo de h após a aplicação das operações. Assim, o menor valor possível de H é quando nenhuma das X operações modifica o valor mínimo de h. Analogamente, o maior valor possível de H é quando todas as X operações modificam o valor mínimo de h, que será incrementado em X unidades. Definimos assim o intervalo de valores que H pode assumir:

$$\min_{i \in [1,n]} \{h_i\} \leq H \leq \min_{i \in [1,n]} \{h_i\} + X$$

Percebemos assim que uma iteração ingênua resulta numa abordagem não polinomial, pois a complexidade de tempo seria proporcional a X.

A observação essencial é que se um valor H é viável, todo $H' < H$ também é. Assim, podemos reduzir o espaço de busca pela metade a cada iteração, já que se estamos procurando o valor máximo de H num intervalo $[l, r]$ e sabemos que $\frac{l+r}{2}$ não é viável, basta procurar no intervalo $[l, \frac{l+r}{2} - 1]$. Analogamente, se $\frac{l+r}{2}$ é viável, podemos encontrar um melhor candidato, então continuamos a procurar no intervalo $[\frac{l+r}{2}, r]$.

Esta abordagem nos dá uma abordagem que explora apenas uma quantidade polinomial de candidatos, proporcional a $\lg X$.

4.2.1 Algoritmo

Utilizando ambas as ideias desenolvidas anteriormente, podemos desenvolver o seguinte algoritmo:

Algoritmo 4.1 Solução para o Problema 4

```
1:  função VALIDO(h, n, X, D, H)
2:      para i de 1 até n faça
3:          dif ← max(H − h[i], 0)
4:          X ← X − dif
5:          para j de i até min(i + D, n) faça
6:              h[j] ← h[j] + dif
7:      se X ≥ 0 então
8:          devolve True
9:      senão
10:         devolve False
11: função RESOLVE(h, n, X, D)
12:     esq ← min(h)
13:     dir ← min(h) + X
14:     enquanto esq < dir
15:         meio ← (esq+dir)/2
16:         se VALIDO(h, n, X, D, meio) então
17:             esq ← meio
18:         senão
19:             dir ← meio − 1
20:     devolve esq
```

No pior caso, a função VALIDO aplica exatamente uma operação em um intervalo com início em cada elemento. Cada operação gasta $O(D)$ para atualizar os elementos de h afetados. Assim, a complexidade de VALIDO é $O(nD)$.

A função RESOLVE, por sua vez, chama VALIDO e reduz o espaço de busca de H na metade a cada iteração. Assim, a complexidade de RESOLVE é $O(nD \lg X)$.

4.2.2 Demonstrações

Provaremos as afirmações feitas na seção anterior. Utilizaremos h^* como notação para o vetor h após a aplicação de um conjunto de operações.

Definiremos H como **viável** se existe um conjunto de até X operações que aplicadas a h produzem um vetor h^* tal que $h^*_i \geq H$ para todo $i \in [1, n]$.

Proposição 4.2. *H é viável $\Rightarrow$ H^* é viável para todo $H^* < H$*

Demonstração. Seja h^* o vetor resultante da aplicação de $k \leq X$ operações necessárias para que $h^*_i \geq H$ para todo $i \in [1, n]$. É trivial que $h^*_i \geq H - 1$. Assim temos que a aplicação das mesmas operações que produzem um vetor h^* no qual $H - 1$ é viável. □

Proposição 4.3. *A função* VALIDO *descrita em 4.1 cria um vetor h^* que respeita a restrição $h^*_i \geq H$ utilizando o menor número de operações possível.*

Demonstração. Definimos uma sequência de operações s como uma sequência de índices ordenados que representam operações num intervalo com tal índice como extremidade esquerda. Por exemplo: $s = (1, 3)$ é a aplicação das operações que incrementam valores de h de índices $[1, min(1 + D, n)]$ e $[3, min(3 + D, n)]$.

Seja s a sequência de operações aplicadas que criam vetor h^* que respeita a restrição de H e seja s' uma sequência de operações tal que $|s'|$ é mínimo, com maior prefixo comum com s que cria um vetor h' que respeita a restrição de H.

Se $s = s'$, está provado.

Caso contrário, seja k o menor índice tal que $s_k \neq s'_k$.

Pela invariante do algoritmo vale que, se $s_i \neq s_{i-1}$, todos os $h^*_j \geq H$ para todo $j < s_i$, ou seja, todos os índices antes de s_i já satisfazem a condição. Também vale que, para todo índice s_i, $h^*_{s_i}$ é justo, isto é, $h^*_{s_i} = H$.

Se $s_k < s'_k$, s' possui uma ocorrência a menos de s_k que s e $h^*_j = H$, então $h'_j < H$, uma contradição na hipótese que h' cria um vetor que respeita a restrição de H.

Se $s_k > s'_k$, s' possui uma ocorrência a mais de s'_k que s. Ora, mas como $h^*_{s'_k} = H$, então h' apresenta folga, ou seja, $h'_{s'_k} > H$. Assim, podemos substituir s'_k por s_k em s', fazendo com que $h'_{s'_k} = H$ e mantendo o mesmo ou aumentando os h'_j com $j > s'_k$. Esta é uma contradição na hipótese que s' tem máximo prefixo comum com s.

□

A função VALIDO descrita em 4.1 utiliza o menor número de operações, se este número for menor ou igual a X, significa que o H fixado é fazível.

4.2.3 Otimizações

Embora o algoritmo 4.1 seja polinomial pois D é limitado por n, ainda há como melhorar a solução. Podemos desconfiar que VALIDO tenha espaço para otimizações.

Precisamos de uma estrutura de dados que dê suporte a incrementar intervalos e acessar posições do vetor de forma eficiente. Uma opção é utilizar uma Árvore de Segmentos com Propagação

Preguiçosa [3], reduzindo a complexidade de cada operação de $O(D)$ para $O(\lg n)$, resultando na complexidade $O(n \lg n \lg X)$.

Podemos atingir uma complexidade ainda melhor que a abordagem anterior sem a utilização de nenhuma estrutura de dados sofisticada.

A observação essencial é estamos iterando em h de maneira sequencial e todas as operações estão sendo feitas em ordem crescente da extremidade esquerda. Deste fato podemos utilizar uma ideia semelhante à Propagação Preguiçosa: manteremos um vetor *preguica* tal que $preguica[i]$ contém quantas operações terminam no índice $i-1$. Com isso, basta manter acumulado numa variável *acum* quantas operações passam pelo índice atual, descontando as operações cujos intervalos não o contém, armazenado em *preguica*. A cada elemento, calculamos quanto falta para que ele seja H, descontando o valor de *acum*. Se for necessário aplicar operações, tanto *acum* quanto *preguica* serão atualizados.

A ideia mostrada pode ser vista no seguinte pseudócodigo:

Algoritmo 4.4 Função VALIDO em tempo linear

```
1:  função ValidoLinear(h, n, X, D, H)
2:      preguica ← {0}^{n+1}
3:      acum ← 0
4:      para i de 1 até n faça
5:          acum ← acum − preguica[i]
6:          dif ← max(H − h[i] − acum, 0)
7:          X ← X − dif
8:          acum ← acum + dif
9:          h[i] ← h[i] + acum
10:         preguica[min(i + D + 1, n + 1)] ← preguica[min(i + D + 1, n + 1)] + dif
11:     se X ≥ 0 então
12:         devolve True
13:     senão
14:         devolve False
```

É fácil ver que ValidoLinear é $O(n)$. Substituindo Valido por ValidoLinear em 4.1 chegamos na complexidade de tempo $O(n \lg X)$ e memória $O(n)$.

4.3 Exercícios

1. Present - Codeforces Round #262 (Div. 2)
2. March Rain - 2016 Al-Baath University Training Camp Contest
3. Assemble - Northwestern Europe Regional Contest 2007
4. Freight Train - Benelux Algorithm Programming Contest 2015
5. Snake Eating - CodeChef SnackDown Online Qualifier 2017

Capítulo 5

Aplicação em Programação Dinâmica

Quando pensamos no escopo de problemas de otimização, pensamos em Programação Dinâmica como uma abordagem completamente disjunta da abordagem gulosa. Neste capítulo mostraremos que, na verdade, é possível aliar ambas as técnicas.

Quando resolvemos problemas como o Problema da Mochila ou o Problema do Troco, fazemos a hipótese de que a ordem em que os itens ou moedas são selecionados não importa, já que estamos interessados apenas no subconjunto final de itens.

A seguir, apresentaremos uma variação do Problema da Mochila onde a ordem em que se colocam os itens na mochila importa e como reduzi-lo ao problema original utilizando uma técnica gulosa.

5.1 Problema da Mochila modificado

Um mochileiro pretende acampar e precisa decidir quais itens colocará em sua mochila e em que ordem, buscando maximizar a soma dos valores destes. Ele escolhe adicionar um item baseado no peso estimado deste e quanto espaço restante há na mochila. Após adicionado, o mochileiro percebe que o item tem peso diferente do estimado, mas não pode mais removê-lo.

Ou seja, existe uma mochila vazia de tamanho S e n itens que desejamos colocar nesta mochila.

Cada item tem dois pesos atrelados a ele: w_i, o peso do item após ser colocado na mochila, e c_i, a estimativa de peso do item antes de ser colocado na mochila. Além disso, cada item tem um valor v_i atrelado a ele.

Um item pode ser colocado na mochila se o espaço restante na mochila é maior que c_i. Após colocado, ele ocupa o peso w_i.

Deseja-se decidir quais itens serão adicionados na mochila e em que ordem, respeitando as restrições de peso de forma que somem o valor máximo.

Formalmente, são dados um inteiro S e três vetores contendo n inteiros positivos cada: $w = \{w_1, w_2, ..., w_n\}$, $v = \{v_1, v_2, ..., v_n\}$, $c = \{c_1, c_2, ..., c_n\}$, onde $c_i \geq w_i$ para todo $i \in [1, n]$. Seja $r = \{r_1, r_2, ..., r_k\}$, $k \leq |I|$, uma permutação de um subconjunto $I \subseteq [1, n]$.

Dizemos que uma permutação r é **válida** se $c_{r_j} + \sum_{i=1}^{j-1} w_{r_i} \leq S$, para todo $j \in [1, k]$, e que um subconjunto I é **válido** se possui uma permutação válida.

Definimos o valor de um subconjunto como:

$$V(I) = \begin{cases} 0 & I \quad \text{não é válido} \\ \sum_{e \in I} v_e & I \quad \text{é válido} \end{cases} \tag{5.1}$$

Dizemos que um subconjunto válido I é ótimo se $V(I)$ é máximo. Desejamos encontrar um subconjunto ótimo I.

Exemplos

i) A entrada $S = 5$, $n = 4$, $v = \{1, 2, 3, 4\}$, $c = w = \{1, 2, 2, 4\}$, tem uma solução ótima $I = \{1, 2, 3\}$, $r = \{1, 2, 3\}$, $V(I) = 1 + 2 + 3 = 6$. Vale notar que $r = \{3, 1, 2\}$ e $r = \{3, 2, 1\}$ também são permutações válidas de I de mesmo valor.

ii) A entrada $S = 5$, $n = 4$, $v = \{1, 2, 3, 4\}$, $c = \{1, 2, 5, 5\}$, $w = \{1, 0, 4, 5\}$, tem uma solução ótima, $I = \{1, 2, 3\}$, $r = \{2, 3, 1\}$, $V(I) = 1 + 2 + 3 = 6$. Desta vez, porém, a ordem importa, já que $r = \{1, 2, 3\}$ e $r = \{3, 2, 1\}$ infringem a condição dos pesos, sendo permutações inválidas de I.

5.2 Desenvolvimento

Primeiramente é necessário observar que se $c = w$, o problema é uma instância do Problema da Mochila clássico. Este, por sua vez, pode ser resolvido com um algoritmo de Programação Dinâmica [9] baseado na recorrência:

$$f(i, s, v, w, n) = \begin{cases} 0 & \text{se } i = n + 1 \\ max(f(i + 1, s, v, w, n), v_i + f(i + 1, s - w_i, v, w, n)) & \text{se } w_i \leq s \\ f(i + 1, s, v, w, n) & \text{c.c.} \end{cases} \tag{5.2}$$

Onde $f(i, s, v, w, n)$ é o maior valor de uma mochila com s de capacidade e $n - i + 1$ itens disponíveis de valores $v_i, v_{i+1}, ..., v_n$, pesos $w_i, w_{i+1}, ..., w_n$.

Uma ideia inicial para adaptar o algoritmo clássico seria modificar a condição de $w_i \leq s$ para $c_i \leq s$, como mostrado abaixo:

$$g(i, s, v, w, c, n) = \begin{cases} 0 & \text{se } i = n + 1 \\ max(g(i + 1, s, v, w, c, n), v_i + g(i + 1, s - w_i, v, w, c, n)) & \text{se } c_i \leq s \\ g(i + 1, s, v, w, c, n) & \text{c.c.} \end{cases} \tag{5.3}$$

Onde $g(i, s, v, w, c, n)$ é o maior valor de uma mochila com s de capacidade e $n - i + 1$ itens disponíveis de valores $v_i, v_{i+1}, ..., v_n$, pesos após colocados na mochila $w_i, w_{i+1}, ..., w_n$ e pesos antes de colocados na mochila $c_i, c_{i+1}, ..., c_n$.

A priori, esta modificação parece suficiente para compreender as restrições impostas pelo enunciado. Esta modificação por si só, porém, não é suficiente.

A recorrência 5.2 assume que, como a ordem de inserção no problema original não importa, é possível fixar uma ordem arbitrária. Neste caso, são os itens do menor ao maior índice, já que i só cresce. Como notado no Exemplo 2, isso não é verdade para este problema. Não basta selecionar o subconjunto de itens a serem inseridos na mochila mas é também necessário encontrar a ordem ótima de inserção destes itens na mochila.

Para lidar com isso, poderíamos aplicar a recorrência para toda escolha de permutação de d, w, v. Isto nos daria uma solucão de complexidade $O(nSn!)$ em tempo, mas é possível fazer melhor que isso.

Podemos desconfiar que existe um critério independente do subconjunto escolhido para encontrar a ordem em que os itens devem ser inseridos e que essa ordem segue algum critério guloso.

Vamos supor que já sabemos o subconjunto ótimo de itens e precisamos apenas decidir sua ordem. Algumas ordens candidatas são:

Decrescente em c

Intuitivamente, essa ordem representa inserir o elemento que é mais pesado antes de ser colocado na mochila o quanto antes. Essa ordem é, porém, rapidamente descartada através do exemplo ii):

A entrada $S = 5$, $n = 4$, $v = \{1, 2, 3, 4\}$, $c = \{1, 2, 5, 5\}$, $w = \{1, 0, 4, 5\}$, tem uma solução ótima na qual $I = \{1, 2, 3\}$ e $r = \{2, 3, 1\}$. A permutação $r = \{3, 2, 1\}$, embora seja decrescente em c, não é válida.

Decrescente em $c - w$

Intuitivamente, essa ordem representa inserir o mais cedo possível itens que têm a maior diferença de peso antes e depois de serem colocados, "aproveitando" quando a mochila tem espaço para satisfazer a condição anterior a inserção, ocupando comparativamente menos espaço após colocá-los.

Essa é, de fato, a ordem correta. Utilizaremos para demonstração um argumento semelhante ao apresentado no capítulo 2.

5.2.1 Algoritmo

A ordenação gulosa pode ser feita com a modificação de um algoritmo de ordenação com complexidade de tempo $O(n \lg n)$ e memória adicional $O(1)$.

A recorrência 5.3 pode ser resolvida com a modificação da Programação Dinâmica para o Problema da Mochila, que apresenta complexidade de tempo $O(nS)$ e de espaço, $O(S)$.

O algoritmo apresenta complexidade tempo $O(n \lg n + nS)$ e memória $O(S)$ se implementado iterativamente e $O(nS)$ se implementado recursivamente.

Algoritmo 5.4 Solução para o Problema 5

```
1: função MOCHILA(i, s, v, w, c, n, memo)
2:     se i = n + 1 então
3:         devolve 0
4:     se memo[i][s] = −1 então
5:         memo[i][s] ← Mochila(i + 1, s, v, w, c, n)
6:         se s ≥ c[i] então
7:             memo[i][s] ← max(memo[i][s], v[i] + Mochila(i + 1, s − w[i], v, w, c, n))
8:     devolve memo[i][s]
9: função RESOLVE(v, w, c, S, n)
10:    Ordene(v, w, c) (decrescente em c − w)
11:    memo ← {{−1}^S}^n
12:    devolve MOCHILA(1, S, v, w, c, n, memo)
```

5.2.2 Demonstrações

Definiremos uma função $F(r)$ como o número de pares de índices (i, j), $i \leq j$, tal que $c_{r_i} - w_{r_i} < c_{r_j} - w_{r_j}$.

Teorema 5.5. *Se $I \subseteq [1, n]$ é um subconjunto ótimo de itens e r é uma permutação de I tal que $c_{r_i} - w_{r_i} \geq c_{r_j} - w_{r_j}$ para todo $i \leq j$, então r é uma permutação válida.*

Demonstração. Se $|I| = 1$, está provado. Se $|I| \neq 1$, suponha que r não seja uma permutação válida. Por definição, $F(r) = 0$.

Seja r^* uma permutação válida de I com menor valor de $F(r^*) > 0$. Seja $t \in [1, |I|-1]$ o primeiro índice tal que $c_{r_t^*} - w_{r_t^*} < c_{r_{t+1}^*} - w_{r_{t+1}^*}$.

Seja $S^* = S - \sum_{i=1}^{t-1} w_{r_i^*}$. Para r^* ser uma permutação válida, temos as condições:

$$c_{r_t^*} \leq S^* \tag{5.6}$$

$$c_{r_{t+1}^*} \leq S^* - w_{r_t^*} \tag{5.7}$$

Note que se $c_{r_{t+1}^*} \leq S^* - w_{r_t^*}$. então:

$$c_{r_{t+1}^*} \leq S^* - w_{r_t^*} \leq S^* \tag{5.8}$$

Note também que se $c_{r_{t+1}^*} \leq S^* - w_{r_t^*}$ e $c_{r_t^*} - w_{r_t^*} < c_{r_{t+1}^*} - w_{r_{t+1}^*}$, então:

$$c_{r_t^*} + w_{r_{t+1}^*} < c_{r_{t+1}^*} + w_{r_t^*} \leq S^*$$

$$c_{r_t^*} < S^* - w_{r_{t+1}^*} \tag{5.9}$$

Percebemos que a condição 5.6 equivale à condição 5.8 e que 5.7 equivale à condição 5.9 numa permutação onde r_t^* e $r^*t + 1$ estão trocados. Ou seja, a sequência $\widetilde{r} = \{r_1^*, ..., r_{t-1}^*, r_{t+1}^*, r_t^*, r_{t+2}^*, ..., r_k^*\}$

é válida e remove pelo menos um par de índices que viola a ordenação gulosa. Assim, temos que $F(\widetilde{r}) < F(r^*)$, uma contradição na hipótese que $F(r^*)$ é mínimo. □

Lema 5.10. *Se d, w, v são ordenados de tal forma que $c_i - w_i \geq c_j - w_j$ para $i \leq j$, então a recorrência 5.3 encontra um subconjunto ótimo.*

Demonstração. Sabemos que a recorrência 5.3 encontra o subconjunto ótimo caso a ordem de inserção dos itens seja a mesma que a ordem crescente dos seus índices. Utilizando a proposição 5.5, para qualquer subconjunto ótimo de d, w, v conforme sua ordenação, existe pelo menos uma permutação válida crescente. Provando assim que a recorrência 5.3 encontra um subconjunto ótimo. □

5.3 Exercícios

Teóricos

1. Suponha que não haja a restrição $c_i \geq w_i$. Prove que o teorema 5.5 continua válido.

Problemas

2. Installing Apps - Northwestern Europe Regional Contest 2017
3. Better Productivity - Northwestern Europe Regional Contest 2015
4. Pope's work - 2011 USP Try-outs

Capítulo 6

Explorando restrições

6.1 Problema da Partição modificado

É dado um vetor de n números inteiros, $V = \{v_1, v_2, ..., v_n\}$, tal que, para todo $i \in [1, n]$, vale que $1 \leq v_i \leq i$.

O objetivo é encontrar, se existir, uma partição de v em dois conjuntos de igual soma.

Alternativamente, queremos encontrar $r = \{r_1, r_2, ..., r_n\}$, com $r_i \in \{-1, 1\}$ para todo $i \in [1, n]$, tal que:

$$\sum_{i=1}^{n} v_i * r_i = 0$$

Exemplos

i) A entrada $n = 4, V = \{1, 2, 3, 3\}$ não apresenta particionamento possível pois $1+2+3+3 = 9$ é ímpar.

ii) A entrada $n = 4, V = \{1, 2, 3, 4\}$ apresenta o particionamento $r = \{-1, 1, 1, -1\}$. Note que o particionamento $r = \{1, -1, -1, 1\}$ também é válido, por simetria.

iii) A entrada $n = 4, V = \{1, 1, 3, 3\}$ apresenta tanto o particionamento $r = \{-1, 1, -1, 1\}$ quanto $r = \{-1, 1, 1, -1\}$, ou seja, podem existir mútiplos particionamentos válidos, não necessáriamente simétricos.

iv) A entrada $n = 3, V = \{1, 1, 3\}$ não apresenta particionamento possível, pois $v_3 = 3 > 1 + 1$.

v) A entrada $n = 3, V = \{1, 2, 4\}$ não é uma entrada válida já que não vale a restrição $1 \leq v_3 \leq 3$.

6.2 Desenvolvimento

Antes de resolver o problema, vale tentar estabelecer conexão com problemas famosos ou de estrutura semelhante.

O primeiro pensamento que vem à tona é a semelhança do enunciado com o Problema da Partição. Na verdade, o problema enunciado pode ser encarado como uma instância particular deste problema clássico.

Isto já nos traz um grande arsenal de informações, pois sabemos que o problema original é NP-completo [6] e pode ser resolvido em tempo pseudo-polinomial usando Programação Dinâmica.

Para obter algoritmos eficientes para o problema devemos, de alguma forma, explorar as restrições adicionais que foram incluídas.

Neste caso, a restrição de entrada $1 \leq v_i \leq i$ é a única diferença entre o Problema da Partição e o problema apresentado. Problemas NP-completo são, à primeira vista, bastante atraentes para soluções gulosas. Evidentemente, um algoritmo baseado em uma estratégia gulosa não produz uma solução ótima para todas as instâncias do problema, pois, como tais algoritmos usualmente têm complexidade polinomial, levaria a uma prova de que P = NP.

Um algoritmo guloso natural é, iterativamente, contruir as partições, começando com duas partições vazias, encaminhamos um elemento à partição com menor soma parcial, tentando deixar as partições mais "equilibradas" possível a todo momento, isto é, minizando a diferença entre as partições em toda iteração.

Além disso, como estamos trabalhando com restrição na entrada em função do índice, temos a intuição de que a ordem em que esses elementos são adicionados importa. Daí seguem duas opções imediatas: iterar crescentemente de 1 a n ou iterar descrescentemente de n a 1.

Iterar crescentemente

Podemos rapidamente descartar esta ordem retornando ao exemplo i) que já analisamos. Na entrada $V = \{1, 2, 3, 4\}$.

Encaminhamos o elemento $v_1 = 1$ para uma partição arbitrária, já que as duas estão vazias. A partir daí, encaminhamos $v_2 = 2$ para a partição oposta a de v_1. Aplicamos o mesmo raciocínio para v_3 e v_4.

Chegando assim na resposta inválida $r = \{-1, 1, -1, 1\}$, que resulta em partições de somas distintas, já que $(-1) * 1 + (+1) * 2 + (-1) * 3 + (+1) * 4 \neq 0$.

Esta instância mostra que o algoritmo não funciona para todos os casos.

Iterar decrescentemente

Vamos testar com a mesma entrada $V = \{1, 2, 3, 4\}$, temos:

Encaminhamos o elemento $v_4 = 4$ para uma partição arbitrária, já que as duas estão vazias. A partir daí, encaminhamos $v_3 = 3$ para a partição oposta a de v_4. Por sua vez, $v_2 = 2$ é encaminhado para a partição de v_3. Aplicamos o mesmo raciocínio para v_1.

Chegando assim na resposta válida $r = \{-1, 1, 1, -1\}$, que resulta em partições de somas iguais, já que $(-1) * 1 + (+1) * 2 + (+1) * 3 + (-1) * 4 = 0$.

Mesmo que exemplos não garantam a corretude da abordagem para todos os casos, realizá-los pode aumentar nossa confiança antes de começarmos uma demonstração formal de corretude.

Já que nossa abordagem apresenta uma série de sucessos em exemplos que montamos, é a hora de tentar desenvolver um algoritmo e desmontrá-lo.

6.2.1 Algoritmo

Para implementar a ideia acima, manteremos uma variável $folgaA$ que possui o espaço restante na partição A. A cada iteração, se houver espaço em A para acomodar v_i, encaminha para tal partição, atualizando $folgaA$. Caso contrário, encaminha para partição B:

Algoritmo 6.1 Solução gulosa para o Problema 6

```
1: função SomaVetor(v, n)
2:     soma ← 0
3:     para i de 1 até n faça
4:         soma ← soma + v_i
5:     devolve soma
6: função Resolve(v, n)
7:     soma ← SomaVetor(v, n)
8:     se soma%2 ≠ 0 então
9:         devolve Impossível
10:    folgaA ← soma/2
11:    r ← {0}^n
12:    para i de n até 1 faça
13:        se folgaA > v_i então
14:            folgaA ← folgaA − v_i
15:            r_i ← 1
16:        senão
17:            r_i ← −1
18:    devolve r
```

É fácil ver que o pseudocódigo acima executa em tempo linear em n, utilizando memória adicional constante.

6.2.2 Demonstrações

Sejam A e B as duas partições tal que dizemos que v_i pertence a partição A se $r_i = 1$ e pertence a B se $r_i = -1$. Seguem algumas definições:

Definição 6.2. *S_i é a soma acumulada do prefixo $v_1, v_2, ..., v_i$ de V: $S_i = \sum_{j=1}^{i} v_j$.*

Definição 6.3. *a_i é a "folga" de A, isto é, a diferença entre o valor esperado da partição e a soma dos elementos que já foram encaminhados a A, depois do processamento dos elementos $v_{i+1}, v_{i+2}, ..., v_n$ pelo algoritmo. Analogamente, definimos b_i para a partição B.*

Proposição 6.4. *$S_n = \sum_{i=1}^{n} v_i$ é impar $\Rightarrow$ Não há solução.*

Proposição 6.5. *Se S_n for par, para qualquer iteração $n-i+1$ do algoritmo, existe folga de tamanho pelo menos v_i em A ou B. Ou seja, $a_i \geq v_i$ ou $b_i \geq v_i$.*

Demonstração. Considere a iteração que decide para que partição v_i será encaminhado, com as partições A e B com folgas a_i e b_i, respectivamente.

Já que todo elemento deverá ser encaminhado para alguma partição, vale que:

$$a_i + b_i = S_i \tag{6.6}$$

Pela definição, sabemos que $i \geq v_i \geq 1$. A partir disso, temos que:

$$S_{i-1} = \sum_{j=1}^{i-1} v_i \geq \sum_{j=1}^{i-1} 1 = i - 1 \geq v_i - 1$$

Assim, podemos escrever S_i como:

$$S_i = S_{i-1} + v_i \geq 2 * v_i - 1 \tag{6.7}$$

Vamos assumir que nem A, nem B, tenham "folga" suficiente para acomodar v_i. Provaremos por contradição, separado em 2 casos:

Caso 1: a_i e b_i têm a mesma paridade.

Se a_i e b_i têm a mesma paridade, S_i é par.

Como nenhuma partição tem "folga" para acomodar v_i, vale que $a_i < v_i$ e $b_i < v_i$. Somando as equações, vale que $a_i + b_i < 2 * v_i$ ou também $a_i + b_i \leq 2 * v_i - 1$.

Utilizando 6.6 e 6.7, segue que $a_i + b_i \geq 2 * v_i - 1$.

Ora mas se $a_i + b_i \geq 2 * v_i - 1$ e $a_i + b_i \leq 2 * v_i - 1$, vale que:

$$S_i = a_i + b_i = 2 * v_i - 1$$

Contradição, já que $2 * v_i - 1$ é ímpar e S_i é par.

Caso 2: a_i e b_i têm diferentes paridades.

Como nenhuma partição tem "folga" para acomodar v_i, temos que $a_i < v_i$ e $b_i < v_i$.

Ora mas como a_i e b_i têm paridades diferentes, deve valer que $a_i < v_i$ e $b_i < v_i - 1$ ou $a_i < v_i - 1$ e $b_i < v_i$.

Vamos assumir, sem perda de generalidade, que $a_i < v_i$ e $b_i < v_i - 1$. Somando ambas as desigualdades, segue que $a_i + b_i < 2 * v_i - 1$ ou $a_i + b_i \leq 2 * v_i - 2$.

Utilizando 6.6 e 6.7, temos que $a_i + b_i \geq 2 * v_i - 1$.

Assim, se $a_i + b_i \geq 2 * v_i - 1$ e $a_i + b_i \leq 2 * v_i - 2$, há uma contradição, já que a intersecção é vazia.

□

6.3 Considerações finais

Vários corolários interessantes saem desta demonstração.

Corolário 6.8. *S_n é par $\Leftrightarrow$ Há solução.*

Demonstração. ($\Leftarrow$) Vide 6.4.

($\Rightarrow$) Estabelecemos na seção anterior um algoritmo que sempre encontra solução caso S_n for par. Assim, S_n ser par é condição suficiente para existência da solução.

□

Corolário 6.9. *Se $a_i > v_i$ e $b_i > v_i$, então v_i pode ser encaminhado para qualquer partição.*

Note que, tanto na construção do algoritmo 6.1 quanto na demonstração da proposição 6.5, não fizemos uso da ideia de encaminhar para a partição com menor soma parcial (ou maior folga parcial), como discutida no início da seção. Tal ideia, porém, dá um critério para decidir qual partição o elemento deve ir, já que a maior folga parcial é sempre positiva.

No algoritmo desenvolvido existe a prioridade de sempre encaminhar o elemento para partição A e, caso não houver "folga" suficiente, envia para partição B. Esta escolha foi, todavia, arbitrária, já que um outro algoritmo não intuitivo que respeita a proposição seria encaminhar o elemento aleatoriamente entre as partições com "folga" de tamanho pelo menos v_i, para todo i.

6.4 Exercícios

1. Hell on the Markets - Northeastern European Regional Contest 2008
2. Painting Eggs - Codeforces Round #173 (Div. 2)
3. Hyper Box - Asia Dhaka Regional Contest 2010

Capítulo 7

Parte Subjetiva

A minha primeira memória de um computador é de quando eu tinha 8 anos de idade. Era 2005, meu pai entrou em casa com um computador munido do poderoso Pentium III, com Windows XP instalado e um jogo, naquela época distribuído em CD-ROM.

Ao ver o computador inicializar pela primeira vez, sabia que era aquilo que eu queria fazer, todo dia, pelo resto da minha vida. Para mim, porém, não bastava jogar, navegar na internet e mandar e-mails: eu sonhava em entender como aquela "caixa preta" funcionava de verdade.

Foram muitas tentativas de me aprofundar no assunto. Depois de abandonar dois cursos técnicos, ser recusado de um curso profissionalizante por ser muito novo e me frustrar com tutoriais na internet, percebi que todos tentaram me ensinar "como fazer" mas não "porque funciona". Assim, ficou claro que precisava buscar ensino superior.

Na faculdade, porém, não seria tão simples quanto eu imaginava. As aulas e os livros muitas vezes não eram suficientes para internalizar completamente os conceitos.

Como um jeito de me aprofundar e continuar evoluindo em algoritmos e estruturas de dados, entrei para o grupo de extensão de estudo para competições de programação, o MaratonIME.

Durante minha participação nessas competições, tive a chance de conhecer e aprender com pessoas muito talentosas, viajar dentro e fora do Brasil, e desenvolver uma capacidade de resolução de problemas que raramente é explorada nos cursos básicos da graduação.

Uma dificuldade, porém, persistia: algoritmos gulosos, o objeto de estudo deste trabalho. Vendo que a dificuldade neste tópico era presente desde estudantes iniciantes no curso até competidores experientes, decidi usar minha experiência de algoritmos "mão na massa" como base da escrita para produzir este material didático.

Espero que tenha sido capaz de transmitir meu conhecimento e contribuído para que mais pessoas consigam aprender computação de maneira mais didática e interessante.

Referências Bibliográficas

[1] Allan Borodin, Morten N. Nielsen, and Charles Rackoff. (incremental) priority algorithms. *Algorithmica*, 37(4):295–326, Dec 2003.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 3rd edition, 2009.

[3] Matheus de Mello Santos Oliveira. Árvores de segmentos. https://linux.ime.usp.br/~matheusmso/mac0499/monografia.pdf, 2018. Acessado em 11 de Dezembro de 2018.

[4] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1(1):127–136, Dec 1971.

[5] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[6] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[7] Eugene L Lawler. *Combinatorial optimization: networks and matroids*. Courier Corporation, 2001.

[8] Dinesh P Mehta and Sartaj Sahni. *Handbook of data structures and applications*. Chapman and Hall/CRC, 2005.

[9] Stefano Tommasini. Programação dinâmica. https://bcc.ime.usp.br/tccs/2014/stefanot/template.pdf, 2014. Acessado em 24 de Novembro de 2018.