Algoritmos Gulosos

Algoritmos para **problemas de otimização geralmente passam por uma sequência de etapas, com um conjunto de escolhas em cada etapa**. Para muitos problemas de otimização, usar programação dinâmica para determinar as melhores escolhas é um exagero; algoritmos mais simples e eficientes servirão.

Um algoritmo guloso, ou ganancioso, sempre faz a escolha que parece melhor no momento. Ou seja, ele **faz uma escolha localmente ótima** na esperança de que essa escolha leve a uma solução globalmente ótima.

Algoritmos gulosos nem sempre produzem soluções ótimas, mas para muitos problemas eles produzem. O método guloso é bastante poderoso e funciona bem para uma ampla gama de problemas.

Como podemos determinar que um algoritmo guloso resolverá um problema de otimização específico?

Nenhuma maneira funciona o tempo todo, mas a propriedade de escolha gananciosa e a subestrutura ideal são os dois ingredientes principais. Se pudermos demonstrar que o problema tem essas propriedades, estaremos no caminho certo para desenvolver um algoritmo guloso para ele.

De maneira mais geral, projetamos algoritmos gulosos de acordo com a seguinte sequência de etapas:

1. Façamos uma escolha de forma a ficarmos com um subproblema para resolver.
2. Prove que há sempre uma solução ótima para o problema original que faz a escolha gulosa, de modo que a escolha gulosa é sempre segura.
3. Demonstre a subestrutura ótima mostrando que, tendo feito a escolha gulosa, o que resta é um subproblema com a propriedade de que se combinarmos uma solução ótima para o subproblema com a escolha gulosa que fizemos, chegaremos a uma solução ótima para o problema original.

Propriedade da Escolha Gulosa

Fazemos a escolha que parece melhor no problema atual, sem considerar os resultados dos subproblemas.

Aqui é onde os algoritmos gananciosos diferem da programação dinâmica. Na programação dinâmica, fazemos uma escolha em cada etapa, mas a escolha geralmente depende das soluções dos subproblemas. Consequentemente, normalmente resolvemos problemas de programação dinâmica de maneira ascendente, progredindo de subproblemas menores para subproblemas maiores.

Diferenciando de Programação Dinâmica

Um algoritmo de programação dinâmica procede de baixo para cima, enquanto uma estratégia gananciosa geralmente progride de cima para baixo, fazendo uma escolha gananciosa após a outra, reduzindo cada instância do problema a uma menor.

Mas claro que devemos provar que uma escolha gananciosa em cada etapa produz uma solução globalmente ótima.

Problema da Seleção de Atividade

Nosso primeiro exemplo é o problema de agendar diversas atividades concorrentes que exigem o uso exclusivo de um recurso comum, com o objetivo de selecionar um conjunto de tamanho máximo de atividades mutuamente compatíveis.

Suponha que temos um conjunto S = {a1, a2 , ... an} de n atividades propostas que desejam usar um recurso, como uma sala de aula, que pode atender apenas a uma atividade por vez.

Cada atividade a_i tem um tempo de início s_i e um tempo de término f_i , onde $0 \leq s_i < f_i < \infty$. Se selecionada, a atividade ai ocorre durante o intervalo de tempo semiaberto $[s_i , f_i)$.

Problema de Seleção de Atividades

No problema de seleção de atividade, desejamos selecionar um subconjunto de tamanho máximo de atividades mutuamente compatíveis. Assumimos que as atividades são classificadas em ordem monotonicamente crescente de tempo de término:

$$f_1 \leq f_2 \leq f_3 \leq ... \leq f_{n-1} \leq f_n \ .$$

Por exemplo, considere o seguinte conjunto S de atividades:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Problema da Seleção de Atividades

Para este exemplo, o subconjunto {a3, a9, a11) consiste em atividades mutuamente compatíveis. Não é um subconjunto máximo, no entanto, já que o subconjunto {a1, a4, a8, a11} é maior. Na verdade, {a1, a4, a8, a11} é um maior subconjunto de atividades mutuamente compatíveis; outro maior subconjunto é {a2, a4, a9, a11}.

A subestrutura ótima

Podemos facilmente verificar que o problema de seleção de atividades exibe uma subestrutura ótima:

Suponha que S_{ij} é o conjunto de atividades que começam após a atividade a_i terminar e que terminam antes da atividade a_j começar. Desejamos encontrar um conjunto máximo de atividades mutuamente compatíveis em S_{ij} , e que tal conjunto máximo seja A_{ij} , que inclui alguma atividade a_k .

Ao incluir a_k em uma solução ótima, ficamos com dois subproblemas:

encontrar atividades mutuamente compatíveis no conjunto S_{ik} e

encontrar atividades mutuamente compatíveis no conjunto S_{kj}.

A subestrutura ótima

Seja

$$A_{ik} = A_{ij} \cap S_{ik} \text{ e}$$

$$A_{kj} = A_{ij} \cap S_{kj}$$

onde,

A_{ik} contém as atividades em A_{ij} que terminam antes de a_k começar e

A_{kj} contém as atividades em A_{ij} que começam depois de a_k terminar.

Assim, temos $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, e assim o conjunto de tamanho máximo A_{ij} de atividades mutuamente compatíveis em S_{ij} consiste em

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1 \text{ atividades.}$$

A subestrutura ótima

Essa maneira de caracterizar a subestrutura ótima sugere que podemos resolver o problema de seleção de atividade por programação dinâmica. Se denotarmos o tamanho de uma solução ótima para o conjunto Sij por c[i, j], então teríamos a recorrência

$$c[i, j] = c[i, k] + c[k, j] + 1$$

Fazendo a escolha gananciosa

A intuição sugere que devemos escolher uma atividade que deixe o recurso disponível para o máximo de outras atividades possível.

Logo uma das atividades que devemos escolher é a primeira a terminar, ou seja, escolher a atividade em S com o tempo de término mais cedo (Se mais de uma atividade em S tiver o tempo de término mais cedo, então podemos escolher qualquer atividade desse tipo.).

Em outras palavras, como as atividades são classificadas em ordem monotonicamente crescente por tempo de término, a escolha gananciosa é a atividade a_1.

Algoritmo Guloso Recursivo

Algoritmo Guloso Recursivo

```
algoritmo seleção-atividades-recursivo(s, f, k, n)
    m = k+1
    enquanto m ≤ n e s[m] < f[k] faça          // encontrar a primeira atividade Sk para encerrar
        m = m+1
    fim_enquanto
    se m ≤ n então
        retorne {am} U seleção-atividades-recursivo(s, f, m, n)
    senão
        retorne ( Φ )
fim_algoritmo
```

Algoritmo guloso iterativo

Algoritmo guloso iterativo

```
algoritmo seleção-atividades( s, f )
    n = s.tamanho
    A = {a1}
    k = 1
    para m = 2 até n
        se s[m] ≥ f[k] então
            A = A U {am}
            k = m
        fim_se
    fim_para
    retorne(A)
fim_algoritmo
```

Código de Huffman

Os códigos de Huffman compactam dados de forma muito eficaz: economias de 20% a 90% são típicas, dependendo das características dos dados que estão sendo compactados.
Consideramos os dados como uma sequência de caracteres. O algoritmo guloso de Huffman usa uma tabela que fornece a **frequência com que cada caractere ocorre** (ou seja, sua frequência) para construir uma maneira ótima de representar cada caractere como uma sequência binária.
Suponha que temos um arquivo de dados de 100.000 caracteres que desejamos armazenar de forma compacta. Observamos que os caracteres no arquivo ocorrem com as frequências dadas pela Figura a seguir. Ou seja, apenas 6 caracteres diferentes aparecem, e o caractere a ocorre 45.000 vezes.

Código de Huffman

	a	b	c	d	e	f
Frequência(em milhares)	45	13	12	19	9	5

Consideramos o problema de **projetar um código de caractere binário (ou código para abreviar) no qual cada caractere é representado por uma sequência binária única**, que chamamos de palavra-código.

Se usarmos um código de comprimento fixo, precisamos de 3 bits para representar 6 caracteres: a = 000, b = 001, . . . , f = 101. Este método requer 300.000 bits para codificar o arquivo inteiro. **Podemos fazer melhor?**

Código de Huffman

Um **código de comprimento variável** pode ter um desempenho consideravelmente melhor do que um código de comprimento fixo, fornecendo palavras-código curtas para caracteres mais frequentes e palavras-código longas para caracteres pouco frequentes.

Códigos de prefixo

Consideramos aqui apenas códigos nos quais nenhuma palavra-código também é um prefixo de alguma outra palavra-código.
Embora não o provemos aqui, um código de prefixo sempre pode atingir a compressão de dados ideal entre qualquer código de caractere e, portanto, não sofremos nenhuma perda de generalidade ao restringir nossa atenção aos códigos de prefixo.

Código prefixo

Códigos de prefixo são desejáveis porque simplificam a decodificação. Como nenhuma palavra-código é um prefixo de qualquer outra, a palavra-código que inicia um arquivo codificado é inequívoca.

O processo de decodificação precisa de uma representação dos códigos de prefixo tal que possamos facilmente identificar a palvra-código inicial. Uma árvore binária cuja as folhas “contém” os caracteres fornecidos é utilizada como representação. Interpretando o caminho da raiz até a folha como sendo a palavra-código, onde o filho da esquerda representa um 0(zero) e o filho da direita representa o 1(hum) no caminho.

Assim, para codificar um conjunto de caracteres é preciso construir a árvore binária com tal representação.

Construindo um código Huffman

Huffman inventou um algoritmo ganancioso que constrói um código de prefixo ótimo chamado código Huffman.

Sua prova de correção depende da propriedade de escolha gananciosa e da subestrutura ótima. Em vez de demonstrar que essas propriedades são válidas e então desenvolver pseudocódigo, apresentamos o pseudocódigo primeiro. Fazer isso ajudará a esclarecer como o algoritmo faz escolhas gananciosas

Código de Huffman

```
algoritmo HUFFMAN(C)
    n = |C|
    Q=C
    para i = 1 até n-1
        crie um novo nó Z
        Z.left = x = EXTRACT-MIN(Q)
        Z.right = y = EXTRACT-MIN(Q)
        Z.freq = x.freq + y.freq
        INSERT(Q,Z)
    fim_para
    retorne(EXTRACT-MIN(Q) //retorna a raiz da árvore
fim_algoritmo
```

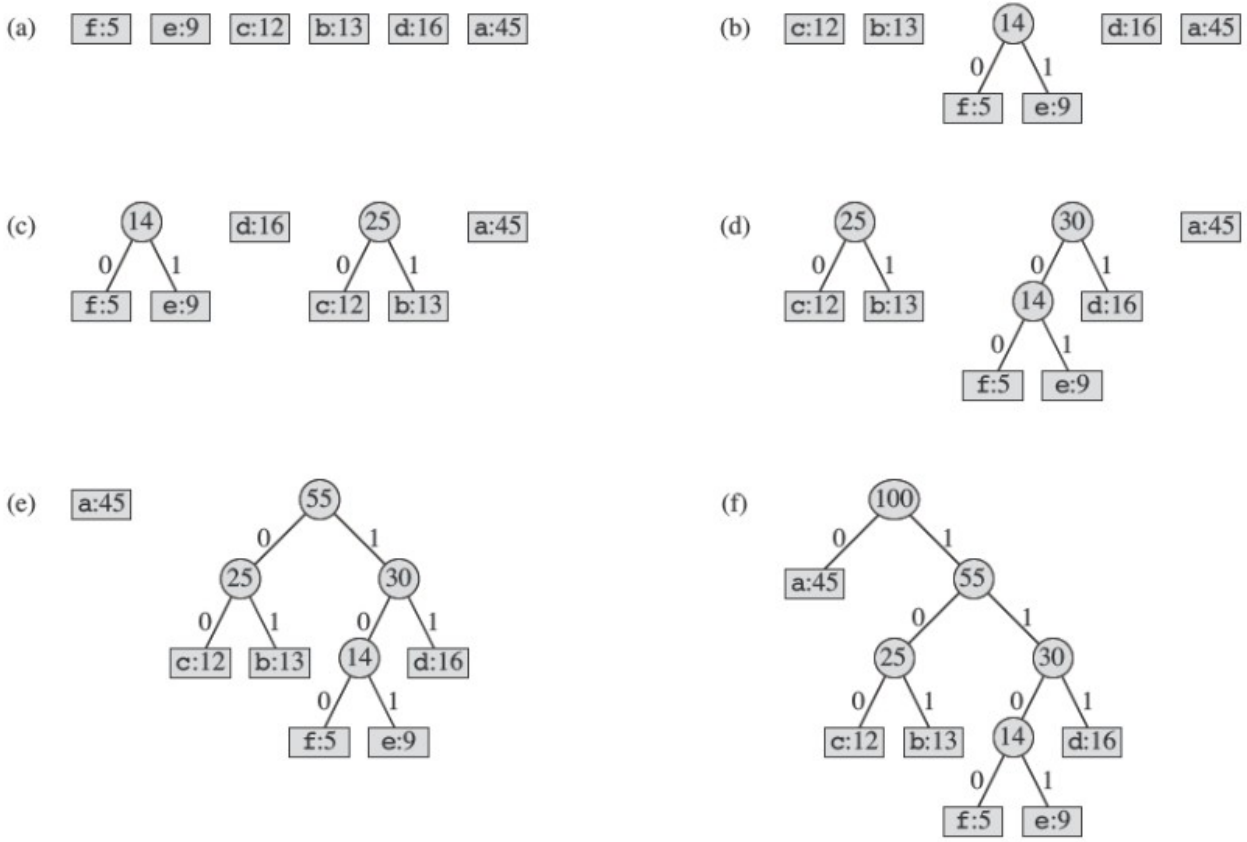
Algoritmo de Huffman

O algoritmo toma como entrada o conjunto de C de caracteres e suas respectivas frequências, faz uma cópia em Q e então passa a construir uma árvore binária de forma ascendente, partindo das folhas até a raiz.

Em cada iteração do laço cria um novo nó interno(Z), escolhendo (e removendo) os dois nós com menor frequência do conjunto, usando a função EXTRACT_MIN(Q), e determina sua frequência como sendo a soma das frequências dos dois escolhidos. Ao final das iterações a árvore binária representando as palavras-códigos é retornada.

A figura a seguir mostra a criação da árvore para a tabela de entrada dada no inicio.

Algoritmo Huffman



Algoritmo de Huffman

A codificação então fica como na seguinte tabela:

	a	b	c	d	e	f
palavras-código	0	101	100	111	1101	1100