Exercício 6.3 (Papadimitriou)

Alice Duarte Scarpa, Bruno Lucian Costa

2015-06-23

1 Enunciado

O Yuckdonald's está considerando abrir uma cadeia de restaurantes em Quaint Valley Highway (QVG). Os n locais possíveis estão em uma linha reta, e as distâncias desses locais até o começo da QVG são, em milhas e em ordem crescente, $m_1, m_2, \ldots, m_n$. As restrições são as seguintes:

- Em cada local, o Yuckdonald's pode abrir no máximo um restaurante. O lucro esperado ao abrir um restaurante no local i é p_i, onde $p_i > 0$ e $i = 1, 2, \ldots, n$.
- Quaisquer dois restaurantes devem estar a pelo menos k milhas de distância, onde k é um inteiro positivo.

Dê um algoritmo eficiente para computar o maior lucro total esperado, sujeito às restrições acima.

2 Introdução

Com este exercício vamos abordar uma técnica chamada de programação dinâmica, que tem como caracteristica que a solução ótima pode ser calculada de soluções de subproblemas.

Antes porém, vai ser apresentado uma solução utilizando um algoritmo guloso.

3 Soluções para o problema

3.1 Algoritmo guloso

Esse algoritmo recebe duas listas de tamanho n, uma com as distâncias dos locais até o ponto inicial e a outra com os respectivos lucros esperados, e um inteiro k que é a distância em milhas do enunciado. Começamos nosso algoritmo saindo do ponto inicial, em direção ao fim da QVH.

Iremos percorrer as lojas em ordem crescente (lembrando que a entrada já vem nessa ordem), escolhendo uma loja sempre que a distância dela for pelo menos o valor da variável `distancia_possivel`. Ao escolhermos uma loja `i`, mudamos o valor dessa variável para `distancia[i] + k`, de modo a não pegar lojas próximas dela.

```python
def restaurante(distancias, k, lucros):
    distancia_possivel = 0
    lucro = 0

    # Percorrendo toda a QVH
    for i in range(len(distancias)):
        if distancias[i] >= distancia_possivel:
            lucro += lucros[i]
            distancia_possivel = distancias[i] + k

    return lucro
```

Vamos mostrar um exemplo no qual esse algoritmo não retonar o valor máximo possivel e vamos tentar entender.

Chamada da função:

```python
print restaurante([3, 8, 9, 15], 3, [5, 6, 10, 8])
```

Resultado:

```
19
```

O resultado obtido utilizando desse algoritmo não foi o resultado ótimo, pois nesse exemplo é fácil perceber que o valor máximo que se pode ter respeitando as restrições é de 23, no qual a escolha seria feita pelos locais $[3, 9, 15]$. O algoritmo guloso, no entanto, está instruído sempre a escolher o primeiro local vago respeitando as restrições, ou seja nesse exemplo ele escolhe os locais $[3, 8, 15]$ totalizando o lucro de 19 que é inferior ao valor ótimo. O algoritmo guloso funcionaria bem para o caso que todos os locais têm o mesmo lucro esperado.

Vamos resolver esse problema com utilizando um algoritmo baseado no paradigma de programação dinâmica.

3.2 Algoritmo utilizando programação dinâmica

Esse técnica de programação utiliza as soluções dos sub-problemas para calcular a solução do problema.

Vamos definir o nosso sub-problema: Suponha $L(i)$ como o lucro máximo que podemos obter com os locais de 1 até i e que $L(0) = 0$. Nosso algoritmo deve seguir a seguinte regra:

$$L(i) = \max(L(i-1), p_i + L(i_{dispo})),$$

onde i_{dispo} é o maior j tal que $m_j \leq m_i - k$, ou seja o primeiro local antes de i que esteja a pelo menos k milhas de distância.

Vamos usar uma função auxiliar `computa_i_dispo` para pré-computar, em tempo linear, o valor de i_{dispo} descrito acima para todo i. A função coloca -1 na posição i do array se não há índice j com essa propriedade, isto é, se $m_i < m_1 + k$ (lembrando que os índices começam de 1 no enunciado, mas que os índices do código começam de zero).

Para tornar o cálculo mais eficiente, exploramos o fato de que a ordenação da entrada implica que $(i+1)_{dispo} \geq i_{dispo}$, isto é, que voltar k milhas a partir da $(i+1)$-ésima casa resulta em um índice maior ou igual que voltar k milhas a partir da i-ésima casa. Isso permite reusar o valor da variável `k_milhas_para_tras` entre iterações do `for`.

```
def calcula_i_dispo(distancias, k):
    i_dispo = []
    k_milhas_para_tras = -1

    for i in xrange(len(distancias)):
        while distancias[k_milhas_para_tras + 1] <= distancias[i] - k:
            k_milhas_para_tras += 1
        i_dispo.append(k_milhas_para_tras)

    return i_dispo
```

O algoritmo acima parece quadrático, mas não é: O loop while interno ou falha o teste e sai imediatamente ou incrementa a variável `k_milhas_para_tras`. A primeira coisa ocorre apenas uma vez por iteração do `for`, e portanto ocorre n vezes no total. A segunda coisa também só ocorre $n-1$ vezes, pois o maior valor possível de `k_milhas_para_tras` é $n-2$, visto que `distancias[n-1]` não pode ser menor que `distancias[i] - k` para nenhum i pois a lista está ordenada.

Usando a função acima, podemos implementar o algoritmo de maneira simples. Esse algoritmo recebe duas listas de tamanho n, uma com as distâncias dos locais até o ponto inicial e a outra com os respectivos lucros esperados, e um inteiro k que é a distância em milhas desejada.

```
def restaurante(distancias, k, valores):
    i_dispo = calcula_i_dispo(distancias, k)

    # Iniciando vetor de zeros de tamanho n+1
    lucros = [0 for _ in range(len(valores) + 1)]

    for i in range(len(distancias)):
        # Calculando L(i_dispo)
        d_novo = lucros[i_dispo[i] + 1]

        # Calculando o lucro acumulado da posicao i
        lucros[i + 1] = max(lucros[i], valores[i] + d_novo)

    return lucros[-1] # retorna o maior lucro calculado
```

Vamos repetir o exemplo que utilizamos no algoritmo guloso e vamos perceber que o valor que retornamos é o valor máximo.

Chamada da função:

```
print restaurante([3, 8, 9, 15], 3, [5, 6, 10, 8])
```

Resultado:

```
23
```

Isso se deve ao fato de que a programação dinâmica utiliza os valores de lucros já calculados e guardados na memória para calcular os valores ótimos futuros.

4 Complexidade

A solução obtida pelo algoritmo guloso possui complexidade linear, porém não é ótima, como podemos perceber no exemplo.

A solução obtida pelo algoritmo utlizando programação dinâmica é a solução ótima e possui complexidade linear, pois a função `calcula_i_dispo` é linear como já argumentado e a função `restaurante` cria um array de tamanho $n + 1$ e depois executa um `for` com operações de tempo constante.