Pedro Henrique Andrade Trindade

(a) Estabeleça uma condição básica para que esse problema admita solução, considerando o vetor F e a distância máxima m que Pudim consegue caminhar sem se alimentar. Essa hipótese será assumida nos itens a seguir.

Resposta:

Para que o problema admita solução, a distância entre quaisquer dois buracos de pesca consecutivos no vetor F não pode exceder m. Formalmente:

$$\forall i \in \{1, \ldots, n-1\}, \quad F[i+1] - F[i] \leq m.$$

Pois se existir um par de buracos consecutivos $(F[i], F[i+1])$ em que a distância $F[i+1] - F[i] > m$, Pudim não consegue ir de um ao outro sem passar fome, tornando o problema insolúvel. Esta condição garante que pelo menos uma solução exista, que é parar em todos os buracos.

(b) Modele uma instância genérica desse problema como um problema de otimização linear inteira. Suponha que sejam dados o vetor F e a distância máxima entre paradas m. Descreva as variáveis de decisão e as restrições adotadas.

Resposta:

As variáveis de decisão para este problema serão definidas de forma que para cada buraco de pesca $i \in \{1, \ldots, n\}$, teremos $x_i \in \mathbb{B}$, onde:

$$x_i = \begin{cases} 1, & \text{se Pudim para no buraco } i, \\ 0, & \text{caso contrário (c.c.)} \end{cases}$$

Nossa função objetivo será definida de maneira a minimar o número total de paradas, não contabilizando a primeira e a última parada, já que sao fixas. Isto é:

$$\text{minimize} \sum_{i=2}^{n-1} x_i.$$

Para as restrições forçaremos por definição que Pudim pare no começo e no final, isto é, $x_1 = 1$ e $x_n = 1$, assim como faremos que a distância máxima entre paradas consecutivas sejam limitadas a m, para garantir que pudim não morra de fome. Esta segunda restrição é um pouco menos trivial, mas conseguimos descrevê-la de maneira linear, para isto seja $B_{>i}(m) = \{k, \text{ tal que } F[k] - F[i] \leq m, \ \forall k > i\}$, isto é, o conjunto de todos os pontos que estão a uma distância menor ou igual a m do ponto i e que estão a frente de i. seja $K_i = \max\{B_{>i}(m)\}$.

Podemos então definir as restrição lineares como:

$$\sum_{k=i+1}^{K_i} x_k \geq x_i, \quad \forall i \in \{1, 2, \ldots, n-1\}$$

Isto se traduz em português para: se x_i for 1 (fizemos uma parada) então deve haver pelo menos algum x_k maior que 1 (significando outra parada) a uma distância menor que m, se $x_i = 0$ note que essa inequação é satisfeita trivialmente.

Formulação Completa:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=2}^{n-2} x_i \\
\text{sujeito a} \quad & \sum_{k=i+1}^{K_i} x_k \geq x_i, \quad \forall i \in \{1, 2, \ldots, n-1\} \\
& x_1 = 1, \\
& x_n = 1, \\
& x_i \in \mathbb{B}, \quad i \in \{1, \ldots, n\}
\end{aligned}$$

[19]:
```
import cvxpy as cp
import numpy as np

def paradas_pudim_pli(F, m):
    n = len(F)
    x = cp.Variable(n, boolean=True)  # Variáveis binárias (1 = parada, 0 = não␣
 ↪parada)

    # Função objetivo: minimizar o número total de paradas
    objective = cp.Minimize(cp.sum(x))

    # Restrições
    constraints = [
        x[0] == 1,   # Obrigatório parar no início
        x[-1] == 1,   # Obrigatório parar no fim
    ]

    # Restrição de distância máxima entre paradas consecutivas
    for i in range(n):
        ball_m = []
        for j in range(i + 1, n):
            if F[j] - F[i] <= m:
                ball_m.append(j)
        if not(ball_m == []):
            constraints.append(cp.sum(x[i+1:np.max(ball_m)+1]) >= x[i])

    # Resolver o problema
```

```python
    prob = cp.Problem(objective, constraints)
    prob.solve(solver=cp.SCIP)  # Solver para problemas inteiros

    if prob.status != cp.OPTIMAL:
        return "Não há solução válida"

    # Extrair as paradas selecionadas
    paradas = [F[i] for i in range(n) if np.isclose(x[i].value, 1.0)]
    return paradas

# Exemplo do enunciado
F = [0, 3, 4, 6, 10, 12]
m = 4
print("Paradas para o Pudim com PLI:")
print(paradas_pudim_pli(F, m))
```

```
Paradas para o Pudim com PLI:
[0, 3, 6, 10, 12]
```

(c) Pudim, além de comilão, é especialista em algoritmos gulosos. A estratégia adotada por Pudim é a seguinte: Pudim irá aguentar o máximo que puder, e só vai parar para pescar se perceber que não conseguirá chegar até o próximo buraco de pesca. Mostre que essa escolha é ótima. Dica: prove por indução.

Resposta:

Vamos utilizar a estratégia gulosa definida em que o Pudim para no buraco mais distante possível dentro do limite m, ou seja, a cada parada i, escolhemos o maior $j > i$ tal que $F[j] - F[i] <= m$.

Vamos provar que qualquer escolha inicial diferente da gulosa é sub-ótima ou equiparavel a estratégia gulosa. Sejam $g_1, g_2, \dots, g_n$ as distância dos buracos em que o algoritmo guloso faz parada e $p_1, p_2, \dots$ para a distância dos buracos em que um algoritmo ótimo faça parada.

Primeiro, escolheremos a base da indução, isso seria tomar $n = 2$, nesse caso o algoritmo guloso é trivialmente ótimo, pois só é possível fazer uma parada, assim como isso é verdade no menor caso em que pode-se haver pelo menos uma parada $n = 3$. (eu fiquei em dúvida se nesse caso eu usava a base ignorando as bordas ou não, por isso coloquei esse segundo pra garantir).

Agora, para o passo indutivo, fazemos o seguinte, suponha que o algoritmo guloso pare em k buracos (não contando o primeiro e o último) e que seja a estratégia ótima, queremos provar que qualquer outra estratégia ótima se equipara para o caso em que o algoritmo guloso faça $k + 1$ paradas.

Note que, pela natureza do problema, temos que p_i tem que estar mais distante do buraco final do que g_i, para qualquer $i \leq k$, em especial vale para p_k e g_k, o que significa que p_{k+1} atingiria no máximo g_{k+1} começando de p_k, então temos que $p_{k+1} \leq g_{k+1}$, portanto, g_{k+1} está mais perto do final, a partir de p_{k+1} teremos que fazer ou mais paradas ou mesmo tanto de paradas do que partindo de g_{k+1}, que chega ao final na próxima iteração, provando o resultado esperado.

[20]:
```python
def paradas_pudim_dp(F, m):
    paradas = [F[0]]  # Sempre começa no primeiro ponto
    ultima_parada = 0  # Índice da última parada
```

```python
    # Enquanto não chegarmos ao final
    while ultima_parada < len(F) - 1:
        # Encontra o buraco mais distante possível dentro do limite m
        prox_parada = ultima_parada
        while (prox_parada + 1 < len(F) and
               F[prox_parada + 1] - F[ultima_parada] <= m):
            prox_parada += 1

        # Se não avançamos, não há solução válida
        if prox_parada == ultima_parada:
            return "Não há solução válida - distância entre buracos muito␣
↪grande"

        # Adiciona a parada encontrada
        paradas.append(F[prox_parada])
        ultima_parada = prox_parada

    return paradas

# Exemplo do enunciado
F = [0, 3, 4, 6, 10, 12]
m = 4

print("Paradas para o pudim com programação dinâmica: ")
print(paradas_pudim_dp(F, m))
```

```
Paradas para o pudim com programação dinâmica:
[0, 4, 6, 10, 12]
```