Pedro Henrique Andrade Trindade

1 Prova Final Programação inteira e Combinacional

1.1 Questão 1: (Arranha-céus!)

(a) Quadrados latinos. Proponha um modelo de otimização linear inteira para resolver o jogo quadrados latinos n × n. Suponha, para o seu modelo, que algumas posições do quadrado estão preenchidas inicialmente. Indique claramente as variáveis de decisão adotadas e justifique as restrições utilizadas.

Resposta:

Para modelar o problema, as variáveis de decisão representarão "qual número será colocado na posição (i, j)".

Como o problema utiliza variáveis discretas, utilizei variáveis de auxílio binárias $z_{ijk} \in \mathbb{B}$ que definem qual valor entre os n possíveis será utilizado naquela posição, qual valor foi utilizado é representado pelo índice k. Assim também foi necessário incluir uma restrição que garanta que apenas um dos números será utilizado por posição, isto é:

$$\sum_{k=1}^{n} z_{ijk} = 1, \ \forall i, j \quad \text{(somente um valor por célula)}$$

Mais explicitamente as variáveis de decisão foram:

$$z_{ijk} \in \{0, 1\} \text{ para } i, j \in \{1, \dots, n\}, k \in \{1, \dots, n\}$$

E para encontrarmos qual valor de fato $x_{ij} \in \{1, 2, \dots, n\}$ se encontrará na posição (i, j) é só uma questão de fazer uma soma ponderada das variáveis de decisão, isto é

$$x_{ij} = \sum_{k=1}^{n} k.z_{ijk}$$

que podemos imaginar como variáveis de decisão a parte e colocar na formulação (como eu fiz) ou pensar na formulação só em função de z_ijk e calcular x_{ij} por fora.

Formulação final:

Para a função objetivo utilizei uma constante dummy C_0

$$
\begin{aligned}
& \text{maximize } C_0 \\
\text{sujeito a } & \sum_{k=1}^{n} z_{ijk} = 1, \quad \forall i,j \quad \text{(somente um valor por célula)} \\
& \sum_{j=1}^{n} z_{ijk} = 1, \quad \forall i,k \quad \text{(cada valor aparece uma vez por linha)} \\
& \sum_{i=1}^{n} z_{ijk} = 1, \quad \forall j,k \quad \text{(cada valor aparece uma vez por coluna)} \\
& x_{ij} = \sum_{k=1}^{n} k \cdot z_{ijk}, \quad \forall i,j \quad \text{(definição do valor final)} \\
& z_{ijk} = 1. \quad \forall (i,j,k) \in \text{(Valores dados das posições fixadas (i,j))}
\end{aligned}
$$

[92]:
```python
import cvxpy as cp
import numpy as np

import warnings
warnings.filterwarnings('ignore')

def make_discrete_decision_variables(array_shape, allowed_values):
    """
    Cria variáveis de decisão binárias para representar valores discretos.
    """
    # Variável binária 3D: z[i,j,k] = 1 se C[i,j] == allowed_values[k]
    z = cp.Variable(array_shape + (len(allowed_values),), boolean=True,␣
 ↪name='z')

    constraints = []
    # Cada célula [i,j] deve ter exatamente um valor selecionado
    for i in range(array_shape[0]):
        for j in range(array_shape[1]):
            constraints += [cp.sum(z[i, j, :]) == 1]  # Somente um valor por␣
 ↪célula

    return z, constraints

def guarantee_one_of_each_in_full_matrix(z_array, allowed_values):
    """
    Garante que cada valor apareça exatamente uma vez em cada linha e coluna.
    """
    constraints = []

    # Para cada linha, cada valor deve aparecer exatamente uma vez
    for row in range(z_array.shape[0]):
        for v_idx, _ in enumerate(allowed_values):
```

```python
            constraints += [cp.sum(z_array[row, :, v_idx]) == 1]

    # Para cada coluna, cada valor deve aparecer exatamente uma vez
    for col in range(z_array.shape[1]):
        for v_idx, _ in enumerate(allowed_values):
            constraints += [cp.sum(z_array[:, col, v_idx]) == 1]

    return constraints

def get_final_solution_constraints(decision_variables, allowed_values):
    """
    Converte as variáveis binárias em uma solução final inteira.
    """
    # Variável final que conterá os valores inteiros
    final_result = cp.Variable(decision_variables[:,:,0].shape, integer=True, name='final')

    # Constrói a matriz resultado combinando as variáveis binárias com os valores permitidos
    final_result_arr = np.zeros(decision_variables[:,:,0].shape)
    for idx in range(decision_variables.shape[2]):
        final_result_arr = final_result_arr + decision_variables[:,:, idx] * allowed_values[idx]

    # Restrição que iguala a variável final ao cálculo
    # (ps: Talvez redundante? Mas a lib não estava me ajudando sem fazer isso)
    constraint = [final_result == final_result_arr]

    return final_result, constraint

# Parâmetros do problema
n = 4
allowed_values = [1, 2, 3, 4]  # Valores discretos permitidos

# Cria as variáveis de decisão e restrições iniciais
decision_variables, constraints = make_discrete_decision_variables((n, n), allowed_values)

# Adiciona restrições de latin square (cada valor uma vez por linha/coluna)
constraints += guarantee_one_of_each_in_full_matrix(decision_variables, allowed_values)

# Prepara a solução final
final_solution, final_solution_constraints = get_final_solution_constraints(decision_variables, allowed_values)
constraints += final_solution_constraints
```

```python
# Restrições de teste para fixar alguns valores
z = decision_variables
"""
Matriz de Teste:
    [[x. 1. x. x.]
     [x. 3. 2. 1.]
     [x. x. x. 3.]
     [x. 4. x. x.]]
"""

test_constraints = [
    z[1, 1, 2] == 1,  # C[1,1] = 3 (allowed_values[2])
    z[1, 2, 1] == 1,  # C[1,2] = 2
    z[1, 3, 0] == 1,  # C[1,3] = 1
    z[2, 3, 2] == 1,  # C[2,3] = 3
    z[3, 1, 3] == 1,  # C[3,1] = 4
    z[0, 1, 0] == 1   # C[0,1] = 1
]
constraints += test_constraints

# Formula e resolve o problema
objective = cp.Maximize(0)
problem = cp.Problem(objective, constraints)
problem.solve(solver=cp.SCIP)

# Exibe os resultados
print("Status da solução:", problem.status)
print("Solução final:")
print(final_solution.value)
```

```
Status da solução: optimal
Solução final:
[[2. 1. 3. 4.]
 [4. 3. 2. 1.]
 [1. 2. 4. 3.]
 [3. 4. 1. 2.]]
```

(b) Prédios vistos em uma fileira. Suponha que uma fileira de n prédios de alturas inteiras e distintas de 1 até n seja descrita pela sequência h1 , · · · , hn das suas alturas. Proponha um modelo de otimização inteira cuja solução forneça o número de prédios que seriam vistos por uma pessoa próxima ao primeiro prédio da fileira. Indique as variáveis adotadas e justifique as restrições utilizadas.

Resposta:

Esta condição aqui foi um pouco mais complicada de pensar, minha linha de raciocínio foi utilizar o fato que podemos representar operações lógicas com PLI.

Assim pensei nas seguintes variáveis de decisão $v_i \in \{1, 2, \dots, n\}$ que define se o prédio i é visível da ponta da fileira, para implementar a lógica por trás dessa definição utilizei variáveis auxiliares $c_{ij} \in \mathbb{B}$, que indicam se o prédio i é tampado pelo prédio j (ou seja, se a altura h_j é maior que h_i, dado que $j < i$, se não c_{ij} trivialmente igual a 0).

Com estas variáveis em mãos, definir se o número de torres visíveis na ponta da fileira é uma questão de fazer o somatório

$$\text{(Número de torres visíveis na borda da fileira)} = \sum_{i=1}^{n} v_i$$

As restrições ficaram um pouco mais convolutas, tentei fazer o seguinte raciocínio

$$\text{(A torre é visível na borda da fileira)} \Leftrightarrow \text{ (Esta torre não é tampada por nenhum } j \text{ anterior)}$$
$$v_i = 1 \Leftrightarrow \sum_j c_{ij} = (i-1), \ \forall (j < i)$$

E também fiz

$$\text{(A torre i não é tampada por j)} \Leftrightarrow (h_i > h_j) \text{ ou } (j \geq i)$$
$$c_{ij} = 1 \Leftrightarrow, (h_j \leq h_i) \text{ e } (j < i)$$

daí foi uma questão de formular estas operações lógicas em termo dessas variáveis booleanas determinadas, utilizando grande M (também precisei usar um *epsilon* de sensibilidade para evitar problemas no simulador).

para o primeiro "se o somente se" as restrições ficaram:

$$\begin{aligned}
&h_i - h_j + \epsilon \leq M.c_{ij} \ \forall i, (j < i) \\
&h_j - h_i - \epsilon \leq M.(1 - c_{ij}) \ \forall i, (j < i) \\
&c_{ij} = 0 \quad \forall i, (j \geq i)
\end{aligned}$$

A ideia aqui é que representamos o "se e somente se" ($\Leftrightarrow$) como duas condições, uma suficiente ($\Rightarrow$) e uma necessária ($\Leftarrow$) que no fundo são representado pelas duas restrições, que forçam o c_{ij} a um ou zero dependo de se a altura de h_i é maior que h_j ou não.

Para o "se e somente se" da visibilidade fiz assim

$$\begin{aligned}
&(i-1) - \sum_{j=1}^{n} c_{ij} + \epsilon \leq M.v_i \ \forall i, \\
&\sum_{j=1}^{n} c_{ij} - (i-1) - \epsilon \leq M.(1 - v_i) \ \forall i,
\end{aligned}$$

Coloquei para função objetivo apenas uma constante dummy C_0, pois satisfazendo-se as equações com as variaveis, já temos nossa resposta como a soma dos v_i.

Formulação Final:

$$
\begin{aligned}
\text{maximize} \quad & C_0 \\
\text{sujeito a} \quad & h_i - h_j + \epsilon \leq M.c_{ij} \quad \forall i, (j < i) \\
& h_j - h_i - \epsilon \leq M.(1 - c_{ij}) \quad \forall i, (j < i) \\
& c_{ij} = 0 \quad \forall i, (j \geq i) \\
& (i-1) - \sum_{j=1}^{n} c_{ij} + \epsilon \leq M.v_i \quad \forall i, \\
& \sum_{j=1}^{n} c_{ij} - (i-1) - \epsilon \leq M.(1 - v_i) \quad \forall i, \\
& c_{ij}, v_i \in \mathbb{B} \quad \forall i, j.
\end{aligned}
$$

[50]:
```python
import cvxpy as cp
import numpy as np
from numpy.typing import NDArray
import random

def get_ordered_comparison_model(line_array: NDArray, epsilon=1e-3, M=1e8):
    """
    Essa função gera um modelo de PLI que podemos utilizar para
    resolver quantas torres são visiveis a partir do primeiro indice
    dado um array de torres.
    """
    # Gera subarrays para comparação
    ordered_list = get_list_of_arrays_less_than_k(line_array)

    # Variável binária que indica as comparações
    Cis = cp.Variable(
        (line_array.shape[0], len(ordered_list)),
        boolean=True,
        name=f'c{random.randrange(1,1000)}'
    ) # Criei um label para as variaveis, para tornar mais fácil debuggar depois

    # Variável que indica se a torre é visível
    tower_visibility = cp.Variable(line_array.shape[0], boolean=True)

    constraints_hi = []

    for idx in range(len(ordered_list)):
        # Restrições do tipo "se e somente se" usando Big-M
        # checando se uma dada torre está sendo tampada por outra anteriormente.
↪
        constraints_hi += [
```

```
            ordered_list[idx][idx] - ordered_list[idx][k] + epsilon <= M *␣
↪(Cis[idx, k])
            for k in range(ordered_list[idx].shape[0])
        ]
        constraints_hi += [
            ordered_list[idx][idx] - ordered_list[idx][k] + epsilon >= -M * (1␣
↪- Cis[idx, k])
            for k in range(ordered_list[idx].shape[0])
        ]

        # Preenche o resto da matriz com zeros
        constraints_hi += [
            Cis[idx, k] == 0
            for k in range(ordered_list[idx].shape[0], len(ordered_list))
        ]

        # Restrições para determinar a visibilidade da torre
        constraints_hi += [
            cp.sum(Cis[idx, :]) - (idx + 1) + epsilon <= M *␣
↪(tower_visibility[idx])
        ]
        constraints_hi += [
            cp.sum(Cis[idx, :]) - (idx + 1) + epsilon >= -M * (1 -␣
↪tower_visibility[idx])
        ]

    return tower_visibility, Cis, constraints_hi

def get_list_of_arrays_less_than_k(line_array: NDArray):
    """
    Gera uma lista de todos os subarrays contendo os primeiros k+1 elementos,
    Para todos os valores menores que o shape do array.
    """
    return [line_array[:k+1] for k in range(line_array.shape[0])]

# Dado de exemplo
A = np.array([3, 2, 4, 1])
print("Subarrays gerados:", get_list_of_arrays_less_than_k(A))

# Parâmetros
epsilon = 1e-5  # Tolerância para desigualdade estrita
M = 1e5         # Valor do Big-M

# Gera o modelo
tower_vis, Cis, constraints = get_ordered_comparison_model(A, epsilon, M)

# Formula o problema de otimização
```

```python
#objective = cp.Maximize(cp.sum(tower_vis))
objective = cp.Maximize(0)

problem = cp.Problem(objective, constraints)

# Resolve o problema
problem.solve(solver=cp.SCIP)

# Imprime resultados
print("Status do problema:", problem.status)

# Resultado final
print(f"Número de torres visíveis: {int(np.sum(tower_vis.value))}")
```

```
Subarrays gerados: [array([3]), array([3, 2]), array([3, 2, 4]), array([3, 2, 4,
1])]
Status do problema: optimal
Número de torres visíveis: 2
```

(c) Modelo Final. Adapte as ideias usadas nos itens anteriores para propor o modelo de otimização inteira que resolve o desafio dos Arranha-Céus. Valide o seu modelo com ao menos 3 exemplos gerados na url sugerida.

Resposta:

Para modelar o problema utilizando as construções feitas anteriormente, foi uma questão de organizar como eu ia representar o jogo. A parte mais complicada foi definir um formato que fosse interessante para as dicas laterais.

A forma como eu fiz foi de fazer um tensor $s_{i\ell m}$ de tamanho $(n, 2, 2)$ onde a primeira entrada representa a posição relativa no quadrado, a segunda coordenada nos fala se a dica é uma dica de linha ou de coluna, já a terceira entrada indica a orientação da dica "direita/esquerda" se for uma dica de linha e "para baixo / para cima" se for uma dica de coluna. Se naquele quadrado não houverem dicas eu defini $s_{ijk} = 0$

Assim iteramos por todas as dicas. Utilizamos as restrições dos quadrados latinos feita anteriormente para o quadrado todo, depois pegamos e adicionamos as restrições de visualização das torres para cada dica, utilizando o problema anterior.

(Ps O conjunto total das restrições ficou relativamente grande, não consegui achar uma forma notacional de representar a formulação toda que fizesse sentido com a mudança de orientações sem apelar para uns negócios chatos, porém é só uma questão sobre reordenar os indices, então vou explicar com palavras)

Para a função objetivo escolhi uma constante dummy C_0. Para cada dica reordenei (por conta da orientação, tipo de dica) as variáveis de entrada nas restrições do problema anterior de acordo, para cada linha eu criei as restrições na ordem normal para todo x_{ij} pertencente aquela linha, para a orientação contrária eu reordenei as variaveis de maneira contrária (e respectivamente na coluna também, só que dessa vez atravessando pelas colunas não linhas) também adicionei a condição de quadrado latino para o quadrado todo (cada x_{ij}).

Em especial adicionei a condição de que o número de torrem visiveis tem que ser igual ao valor da dica para cada dica, isto é $\sum_{i=1}^{n} v_{i\ell} = s_\ell, \ \forall \ell$ (denovo, desculpa pela notação confusa, os indices mudar de orientacao me atrapalharam muito).

[87]:
```python
def get_side_towers_constraints(final_result_vars, side_towers):
    constraints = []

    # Variável que representa a soma total das torres visíveis dado cada␣
 ↪posição, tipo e orientação
    # das pistas laterais
    total_sum = cp.Variable(side_towers.shape, integer=True, name='tower_sums')

    # Percorre todas as posições da matriz de pistas laterais
    for position in range(side_towers.shape[0]):
        for pos_type in range(side_towers.shape[1]):
            for orientation in range(side_towers.shape[2]):
                towers_value = side_towers[position, pos_type, orientation]

                if not (towers_value == 0):
                    # Determina se a visualização será por linha ou por coluna␣
 ↪e em qual direção
                    if pos_type == 0:  # Linha
                        if orientation == 0:  # Da esquerda para a direita
                            tower_vis, _, tower_constraint =␣
 ↪get_ordered_comparison_model(final_result_vars[position, :])
                        else:  # Da direita para a esquerda
                            tower_vis, _, tower_constraint =␣
 ↪get_ordered_comparison_model(final_result_vars[position, ::-1])
                    else:  # Coluna
                        if orientation == 0:  # De cima para baixo
                            tower_vis, _, tower_constraint =␣
 ↪get_ordered_comparison_model(final_result_vars[:, position])
                        else:  # De baixo para cima
                            tower_vis, _, tower_constraint =␣
 ↪get_ordered_comparison_model(final_result_vars[::-1, position])

                    # Adiciona restrições relacionadas à soma das torres␣
 ↪visíveis
                    constraints += [total_sum[position, pos_type, orientation]␣
 ↪== cp.sum(tower_vis)]
                    constraints += tower_constraint
                    constraints += [towers_value == cp.sum(tower_vis)]
                else:
                    # Se não há torre na posição, a soma deve ser zero
                    constraints += [total_sum[position, pos_type, orientation]␣
 ↪== 0]
```

```python
    return total_sum, constraints


def solve_towers_problem(n, side_towers):
    # Lista de restrições inicial da modelagem
    constraints = []

    # Conjunto de valores permitidos
    allowed_values = list(range(1, n+1))  # Conjunto discreto 1..n

    # Cria variáveis de decisão para a matriz do quebra-cabeça
    decision_variables, constraints = make_discrete_decision_variables((n, n),␣
 ↪allowed_values)

    # Garante que cada valor apareça uma vez em cada linha e coluna
    constraints += guarantee_one_of_each_in_full_matrix(decision_variables,␣
 ↪allowed_values)

    # Gera as variáveis finais que representam a solução e suas restrições
    final_solution, final_solution_constraints =␣
 ↪get_final_solution_constraints(decision_variables, allowed_values)
    constraints += final_solution_constraints

    # Aplica as restrições de visualização das torres laterais
    _, tower_constraints = get_side_towers_constraints(final_solution,␣
 ↪side_towers)
    constraints += tower_constraints

    # Define o objetivo da otimização: maximizar a soma das variáveis e da␣
 ↪visualização das torres
    objective = cp.Maximize(0)

    # Cria e resolve o problema de otimização
    problem = cp.Problem(objective, constraints)
    problem.solve(solver=cp.SCIP)

    # Exibe o status da solução e a matriz final encontrada
    print(f"\nStatus da solução: {problem.status}")
    print(f"\nValor do resultado exemplo: \n{final_solution.value}")


# Matriz que define as torres visíveis pelas laterais: [posição, tipo (linha/
 ↪coluna), orientação]
s = np.zeros((n, 2, 2))  # posição, tipo (linha ou coluna), orientação (direita␣
 ↪ou esquerda, baixo ou cima)
```

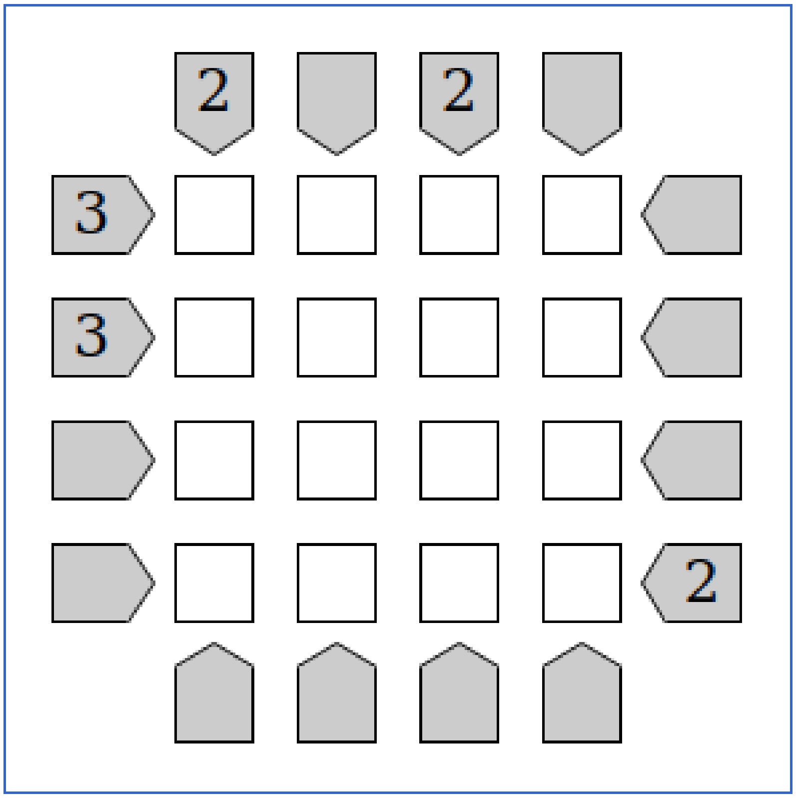
```python
## Teste: define algumas pistas nas laterais, exemplo do enunciado
## Linhas
s[1, 0, 0] = 2  # Linha 1, da esquerda para a direita
s[3, 0, 1] = 4  # Linha 3, da direita para a esquerda

## Colunas
s[1, 1, 0] = 1  # Coluna 1, de cima para baixo
s[0, 1, 0] = 3  # Coluna 0, de cima para baixo
s[3, 1, 1] = 2  # Coluna 3, de baixo para cima

# Tamanho da grade
n = 4

solve_towers_problem(n, s)
```

```
Status da solução: optimal

Valor do resultado exemplo:
[[2. 4. 1. 3.]
 [3. 1. 4. 2.]
 [1. 2. 3. 4.]
 [4. 3. 2. 1.]]
```

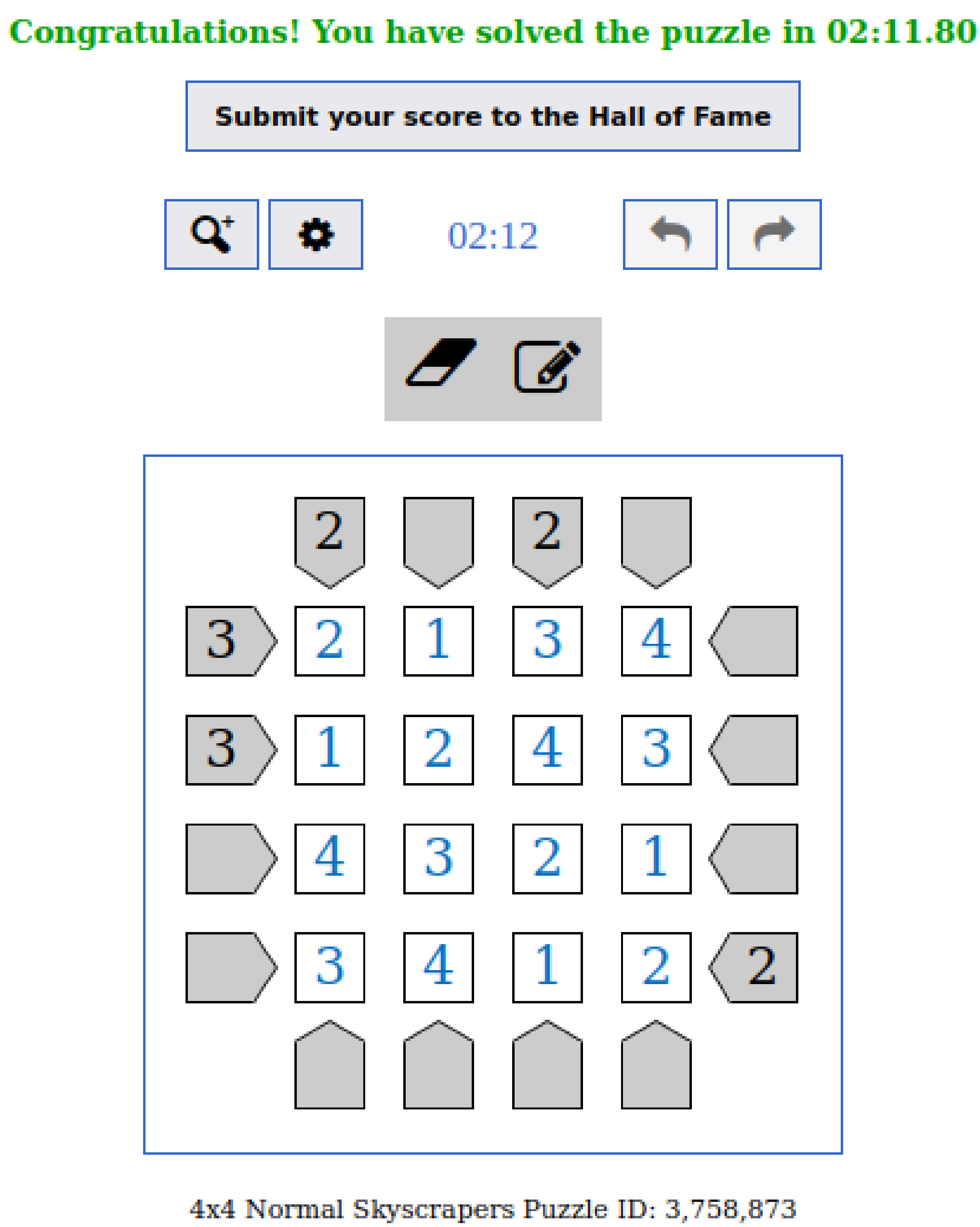
1.2 Exemplos da URL

1.2.1 Exemplo 1

Problema:



Soluçao:



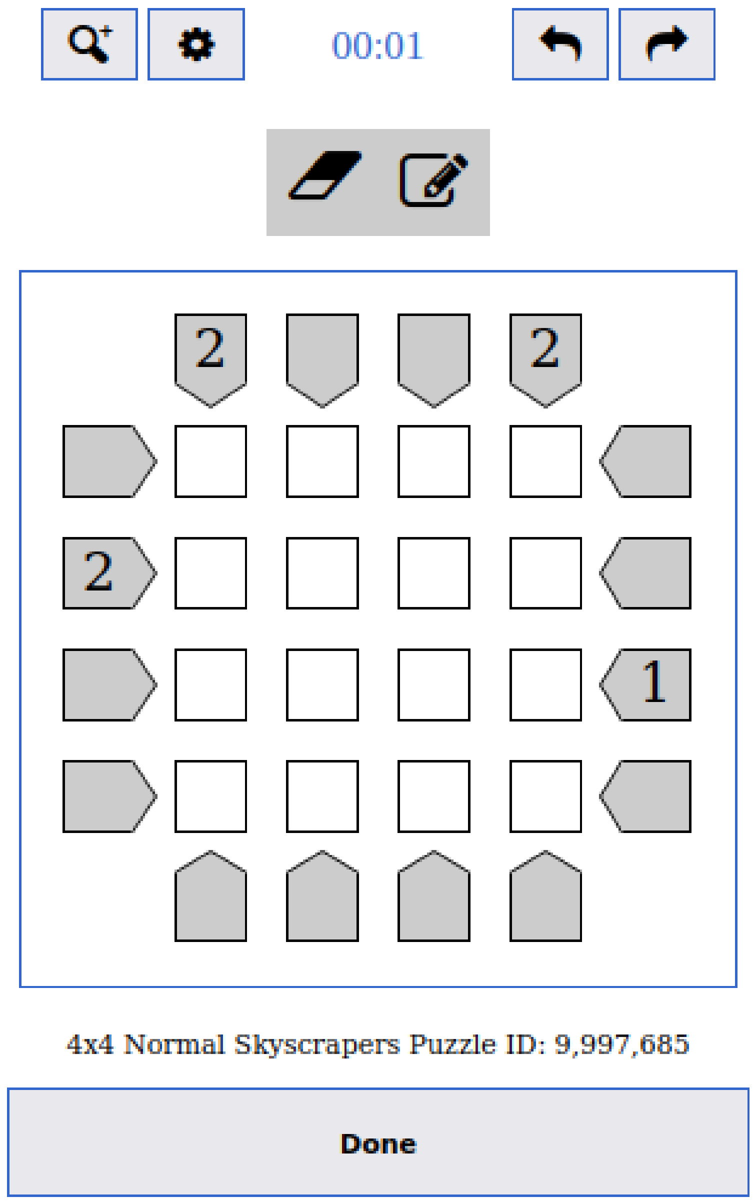
[82]:
```python
# Tamanho da grade
n = 4
# Matriz que define as torres visíveis pelas laterais: [posição, tipo (linha/coluna), orientação]
exemplo1 = np.zeros((n, 2, 2))  # posição, tipo (linha ou coluna), orientação (direita ou esquerda, baixo ou cima)

## Teste: define algumas pistas nas laterais, exemplo do enunciado
exemplo1[0, 0, 0] = 3
```

```
exemplo1[1, 0, 0] = 3
exemplo1[0, 1, 0] = 2
exemplo1[2, 1, 0] = 2
exemplo1[3, 0, 1] = 2

solve_towers_problem(n, exemplo1)
```

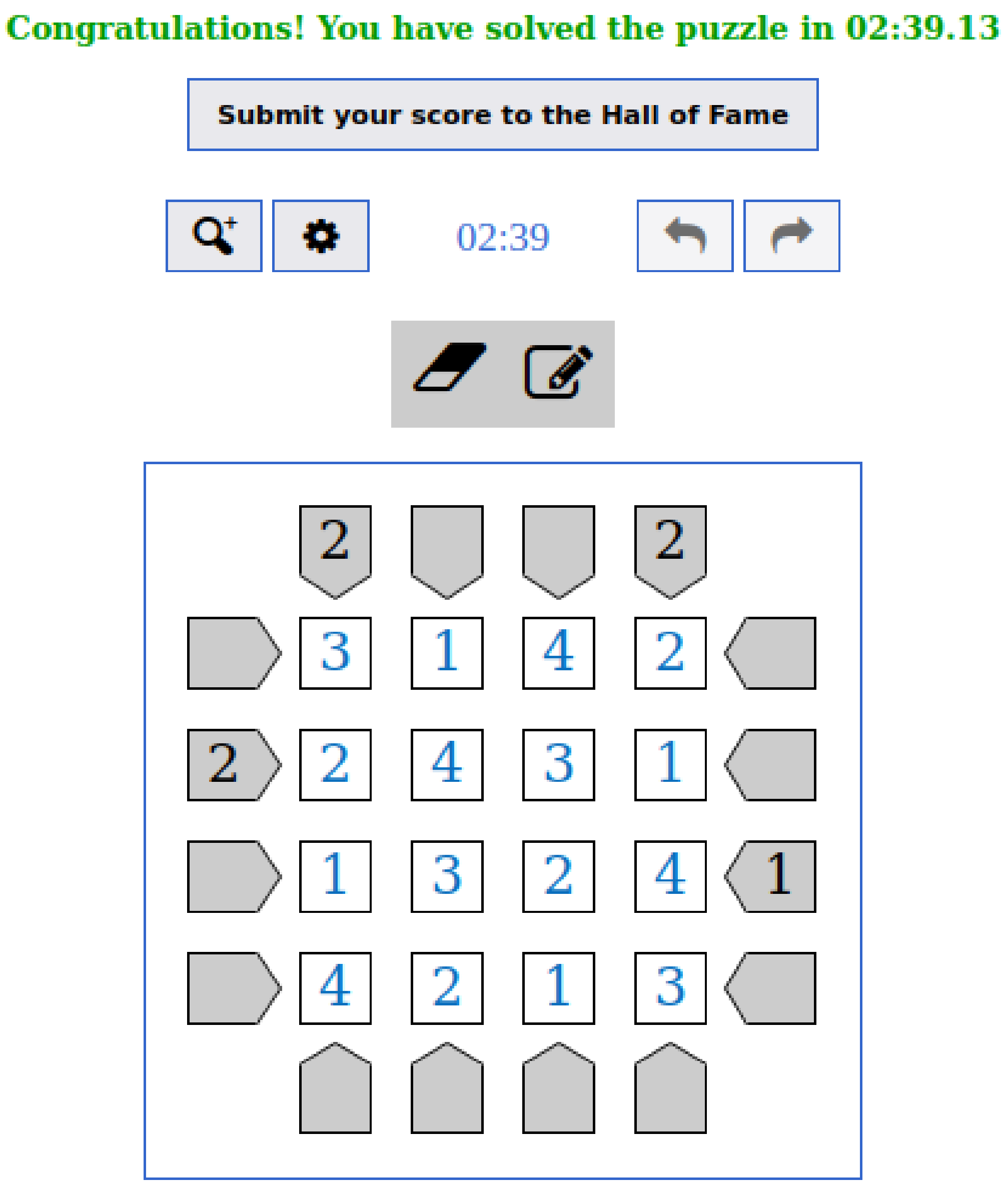
```
Status da solução: optimal

Valor do resultado exemplo:
[[2. 1. 3. 4.]
 [1. 2. 4. 3.]
 [4. 3. 2. 1.]
 [3. 4. 1. 2.]]
```

1.2.2 Exemplo 2

Problema:



Soluçao:



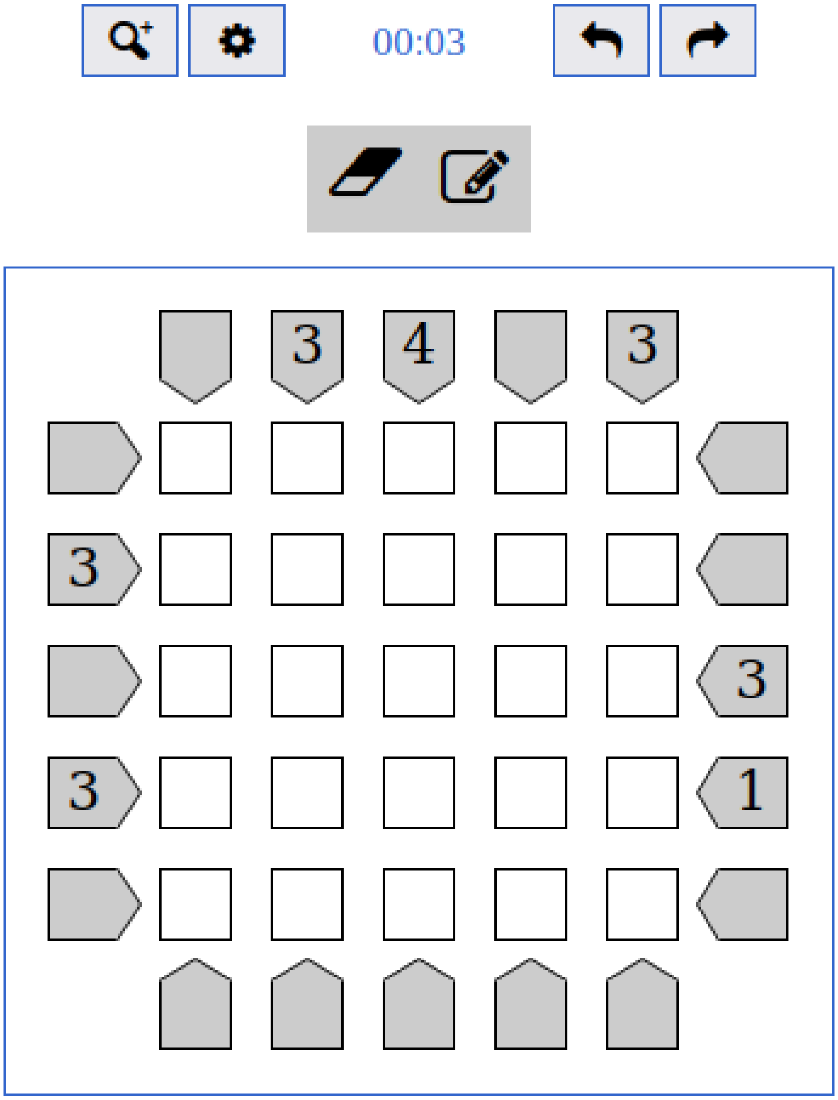
[84]:
```
# Tamanho da grade
n = 4
# Matriz que define as torres visíveis pelas laterais: [posição, tipo (linha/coluna), orientação]
exemplo2 = np.zeros((n, 2, 2))  # posição, tipo (linha ou coluna), orientação (direita ou esquerda, baixo ou cima)

## Teste: define algumas pistas nas laterais, exemplo do enunciado
exemplo2[1, 0, 0] = 2
exemplo2[0, 1, 0] = 2
exemplo2[3, 1, 0] = 2
exemplo2[2, 0, 1] = 1
```

```
solve_towers_problem(n, exemplo2)
```

```
Status da solução: optimal

Valor do resultado exemplo:
[[3. 1. 4. 2.]
 [2. 4. 3. 1.]
 [1. 3. 2. 4.]
 [4. 2. 1. 3.]]
```

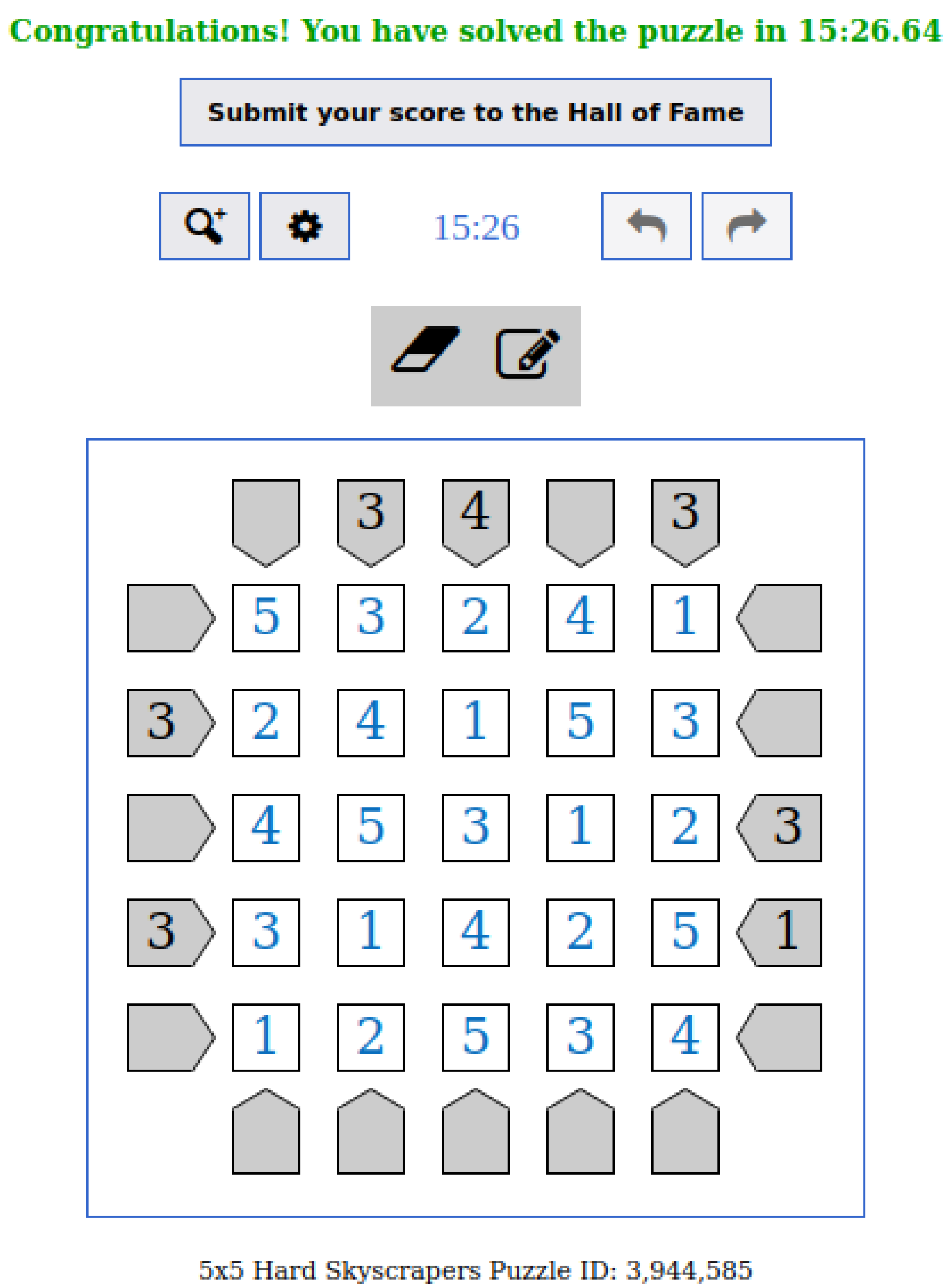
1.2.3 Exemplo 3

Problema:



5x5 Hard Skyscrapers Puzzle ID: 3,944,585

Soluçao:



[80]:
```python
# Tamanho da grade
n = 5
# Matriz que define as torres visíveis pelas laterais: [posição, tipo (linha/coluna), orientação]
exemplo3 = np.zeros((n, 2, 2))  # posição, tipo (linha ou coluna), orientação (direita ou esquerda, baixo ou cima)

## Teste: define algumas pistas nas laterais, exemplo do enunciado
exemplo3[1, 0, 0] = 3
exemplo3[3, 0, 0] = 3
```

```python
exemplo3[1, 1, 0] = 3
exemplo3[2, 1, 0] = 4
exemplo3[4, 1, 0] = 3
exemplo3[2, 0, 1] = 3
exemplo3[3, 0, 1] = 1

solve_towers_problem(n, exemplo3)
```

```
Status da solução: optimal

Valor do resultado exemplo:
[[5. 3. 2. 4. 1.]
 [2. 4. 1. 5. 3.]
 [4. 5. 3. 1. 2.]
 [3. 1. 4. 2. 5.]
 [1. 2. 5. 3. 4.]]
```