Lógica Computacional: 25/26

TP1 - Ex2

Grupo 05

- Vasco Ferreira Leite (A108399)
- Gustavo da Silva Faria (A108575)
- Afonso Henrique Cerqueira Leal (A108472)

Neste trabalho pretende-se generalizar o problema em várias direções:

- **Em primeiro lugar**, a grelha tem como parâmetro fundamental um inteiro que toma vários valores $n \in \{3, 4, \ldots\}$. Fundamentalmente a grelha passa de um quadrado com $n^2 \times n^2$ células para um cubo tridimensional de dimensões $n^2 \times n^2 \times n^2$. Cada posição na grelha é representada por um triplo de inteiros $(i, j, k) \in \{1, \ldots, n^2\}^3$.

- **Em segundo lugar**, as "regiões" que a definição menciona deixam de ser linhas, colunas e "sub-grids" para passar a ser qualquer "box" genérica com um número de células $\leq n^3$. Cada "box" é representada por um dicionário D que associa, no estado inicial, cada posição $(i, j, k) \in D$ na "box" a um valor inteiro no intervalo $\{0, \ldots, n^3\}$.

- Na inicialização da solução, as células associadas ao valor 0 estão livres para ser instanciadas com qualquer valor não nulo. Se nessa fase, uma célula está associada a um valor não nulo, então esse valor está **fixo** e qualquer solução do problema não o modifica.

- A solução final do problema, tal como no problema original, verifica uma restrição do tipo *all-different* que, neste caso, tem a forma:

  > Dentro de uma mesma "box", todas as células têm valores distintos no intervalo $\{1, \ldots, n^3\}$.

- Consideram-se neste problema duas formas básicas de "boxes":

  - **"Cubos"** de n^3 células determinados pelo seu vértice superior, anterior, esquerdo.
  - **"Paths"** determinados pelo seu vértice de início, o vértice final e pela ordem entre os índices dos vértices sucessivos.

O **input** do problema é um conjunto de "boxes" e um conjunto de alocações de valores a células.

Variáveis:

Variáveis de decisão:

- `cells[(i,j,k)]` : Valor da célula na posição (i, j, k), domínio $[1, n^3]$

Parâmetros:

- `n` : Variável base do problema
- `size` : n^2 (dimensão de cada aresta do cubo)
- `domain_size` : n^3 tamanho do domínio (maior valor tomado por uma célula)
- `valoresFixos` : Dicionário das células com valores pré-definidos
- `cube_boxes` : Lista das células que representam as boxes básicas $n \times n \times n$ do sudoku
- `path_boxes` : Lista com os caminhos custumisados e as células que os formam
- `generic_boxes` : Lista de boxes customizadas

In [ ]:
```
from ortools.sat.python import cp_model
```

Função: Criar o modelo

Cria o modelo CP-SAT, inicializa as variáveis para todas as células do cubo, adiciona os valores fixos fornecidos como input, aplica todas as restrições, e invoca o solver para encontrar uma solução.

In [ ]:
```
def solve_sudoku_3d(n, valoresFixos=None, cube_boxes=None, path_box

    size = n * n
    domain_size = n * n * n

    model = cp_model.CpModel()
    solver = cp_model.CpSolver()

    cells = {}
    for i in range(1, size + 1):
        for j in range(1, size + 1):
            for k in range(1, size + 1):
                cells[(i, j, k)] = model.NewIntVar(1, domain_size,

    if valoresFixos:
        for (i, j, k), value in valoresFixos.items():
            if value != 0:
                model.Add(cells[(i, j, k)] == value)
```

```
    _add_basic_constraints(model, cells, size)

    if cube_boxes:
        for start_i, start_j, start_k in cube_boxes:
            _add_cube_box(model, cells, n, size, start_i, start_j, 

    if path_boxes:
        for path in path_boxes:
            path_vars = [cells[cell] for cell in path]
            if len(path_vars) > 1:
                model.AddAllDifferent(path_vars)

    if generic_boxes:
        for box_dict in generic_boxes:
            _add_generic_box(model, cells, box_dict)

    status = solver.Solve(model)

    if status in [cp_model.OPTIMAL, cp_model.FEASIBLE]:
        solution = {}
        for (i, j, k), var in cells.items():
            solution[(i, j, k)] = solver.Value(var)
        return solution
    else:
        print(f"Solução não encontrada. Status: {status}")
        return None
```

Função: Restrições Básicas do Sudoku

Esta função implementa as restrições fundamentais do Sudoku a todas as células.
Para cada uma das três direções:

- **Linhas** (ao longo de `k`)
- **Colunas** (ao longo de `j`)
- **Pilhas** (ao longo de `i`)

Aplica a restrição **`AllDifferent`**, garantindo que não há valores repetidos em nenhuma dessas direções.

In [ ]:
```
def _add_basic_constraints(model, cells, size):

    for i in range(1, size + 1):
        for j in range(1, size + 1):
            row_cells = [cells[(i, j, k)] for k in range(1, size + 
            model.AddAllDifferent(row_cells)

    for i in range(1, size + 1):
        for k in range(1, size + 1):
            col_cells = [cells[(i, j, k)] for j in range(1, size + 
            model.AddAllDifferent(col_cells)
```

```
    for j in range(1, size + 1):
        for k in range(1, size + 1):
            stack_cells = [cells[(i, j, k)] for i in range(1, size +
            model.AddAllDifferent(stack_cells)
```

Função: Restrição box básica

Implementa boxes do tipo cubo, definidas pelo seu vértice superior-anterior-esquerdo. Constrói um subcubo de dimensões $n \times n \times n$ e aplica a restrição AllDifferent a todas as suas n^3 células.

In [ ]:
```
def _add_cube_box(model, cells, n, size, start_i, start_j, start_k)

    cells_in_box = []

    for di in range(n):
        for dj in range(n):
            for dk in range(n):
                i = start_i + di
                j = start_j + dj
                k = start_k + dk
                if 1 <= i <= size and 1 <= j <= size and 1 <= k <= 
                    cells_in_box.append(cells[(i, j, k)])

    if len(cells_in_box) > 1:
        model.AddAllDifferent(cells_in_box)

    return cells_in_box
```

Função: Restrição box custumizada

Implementa boxes genéricas customizáveis através de um dicionário. Permite definir qualquer conjunto de células como uma box e aplica all-diferent nas mesmas, onde células com valor 0 são livres e devem ter valores diferentes entre si, e células com valores não-nulos têm esses valores fixos.

In [ ]:
```
def _add_generic_box(model, cells, box_dict):

    free_cells = []

    for (i, j, k), value in box_dict.items():
        if value == 0:
            free_cells.append(cells[(i, j, k)])
        else:
            model.Add(cells[(i, j, k)] == value)

    if len(free_cells) > 1:
        model.AddAllDifferent(free_cells)

    return free_cells
```

Função: Visualização da sulução

Função auxiliar para visualização da solução 3D. Esta função primeiro imprime os vários "paths", "generic boxes" e os valores associados as células que os formam. De seguida permite visualizar o cubo em camadas ao longo de um eixo.

In [ ]:
```
def print_solution_colored(solution, n, valoresFixos=None, path_cel
    if not solution:
        print("Nenhuma solução para imprimir")
        return

    BLUE = '\033[94m'
    GREEN = '\033[92m'
    BOLD = '\033[1m'
    RESET = '\033[0m'
    GRAY = '\033[90m'
    CYAN = '\033[96m'
    MAGENTA = '\033[95m'

    size = n * n
    valoresFixos = valoresFixos or {}

    if path_cells_list:
        print(f"\n{BOLD}{CYAN}=== PATH BOXES ==={RESET}")
        for idx, path in enumerate(path_cells_list, start=1):
            print(f"\n{BOLD}Path #{idx}:{RESET}")
            values = []
            for cell in path:
                val = solution.get(cell, None)
                if val is not None:
                    values.append(val)
                    print(f"  ({cell[0]:2d},{cell[1]:2d},{cell[2]:2
            if len(values) == len(set(values)):
                print(f"  {CYAN}✔ Todos diferentes{RESET}")
            else:
                print(f"  {MAGENTA}✘ Repetições encontradas!{RESET}"

    if generic_boxes:
        print(f"\n{BOLD}{CYAN}=== GENERIC BOXES ==={RESET}")
        for idx, box in enumerate(generic_boxes, start=1):
            print(f"\n{BOLD}Box #{idx}:{RESET}")
            values = []
            for (i, j, k), fixed in box.items():
                val = solution.get((i, j, k), None)
                if fixed != 0:
                    print(f"  ({i:2d},{j:2d},{k:2d}) = {BLUE}{val}
                    values.append(val)
                else:
                    print(f"  ({i:2d},{j:2d},{k:2d}) = {GREEN}{val}
                    values.append(val)
            if len(values) == len(set(values)):
```

```
                print(f"  {CYAN}✔ Todos diferentes{RESET}")
            else:
                print(f"  {MAGENTA}✘ Repetições encontradas!{RESET}"

    for k in range(1, size + 1):
        print(f"\n{BOLD}╔{'=' * (size * 4 + 1)}╗{RESET}")
        print(f"{BOLD}║ Camada k = {k:2d}{' ' * (size * 4 - 12)}║{R
        print(f"{BOLD}╠{'=' * (size * 4 + 1)}╣{RESET}")

        for i in range(1, size + 1):
            row_str = f"{BOLD}║{RESET} "

            for j in range(1, size + 1):
                value = solution[(i, j, k)]
                is_fixed = (i, j, k) in valoresFixos and valoresFix

                if is_fixed:
                    row_str += f"{BLUE}{BOLD}{value:3d}{RESET} "
                else:
                    row_str += f"{GREEN}{value:3d}{RESET} "

                if j % n == 0 and j < size:
                    row_str += f"{GRAY}│{RESET} "

            row_str += f"{BOLD}║{RESET}"
            print(row_str)

            if i % n == 0 and i < size:
                print(f"{BOLD}║{RESET}{GRAY}{'─' * (size * 4 + 1)}{

        print(f"{BOLD}╚{'=' * (size * 4 + 1)}╝{RESET}")

    print(f"\n{BLUE}■{RESET} Valores fixos (input)  {GREEN}■{RESET}
```

Exemplo de problema

n = 4

In [ ]:

```
n = 4
size = n * n
domain_size = n * n * n
valoresFixos = {}

valoresFixos[(1, 1, 1)] = 1
valoresFixos[(1, 1, 16)] = 16
valoresFixos[(1, 16, 1)] = 32
valoresFixos[(16, 1, 1)] = 48
valoresFixos[(16, 16, 16)] = 64

center = size // 2
quarter = size // 4

valoresFixos[(center, center, center)] = domain_size // 2
valoresFixos[(1, center, center)] = 31
```

```
valoresFixos[(16, center, center)] = 42
valoresFixos[(center, 1, center)] = 44
valoresFixos[(center, 16, center)] = 27
valoresFixos[(center, center, 1)] = 56
valoresFixos[(center, center, 16)] = 38
valoresFixos[(quarter, quarter, quarter)] = 16
valoresFixos[(size-quarter, quarter, quarter)] = 41
valoresFixos[(quarter, size-quarter, quarter)] = 45
valoresFixos[(size-quarter, size-quarter, quarter)] = 11

cube_boxes = []
for start_i in range(1, size + 1, n):
    for start_j in range(1, size + 1, n):
        for start_k in range(1, size + 1, n):
            cube_boxes.append((start_i, start_j, start_k))

path1 = [(t, t, t) for t in range(1, 17)]
path2 = [(t, t, 17 - t) for t in range(1, 17)]
path3 = [(t, 17 - t, t) for t in range(1, 17)]
path4 = [(17 - t, t, t) for t in range(1, 17)]

path_boxes = [path1, path2, path3, path4]

generic_box1 = {
    (1, 2, 2): 0,
    (3, 4, 4): 0,
    (4, 3, 3): 5,
    (5, 4, 4): 0,
    (6, 5, 5): 0,
    (7, 6, 6): 0,
}

generic_boxes = [generic_box1]
```

Execução

In [ ]:

```
solution = solve_sudoku_3d(
    n=n,
    valoresFixos=valoresFixos,
    cube_boxes=cube_boxes,
    path_boxes=path_boxes,
    generic_boxes=generic_boxes
)

if solution:
    print("Solução encontrada!")
    print_solution_colored(solution, n, valoresFixos, path_boxes, g
else:
    print("\nx Não foi possível encontrar solução")
```