Formulaire B&B

========== **Shortest Path** ==========

```python
class Cell:
    def __init__(self, coords: Tuple[int, int], board: List[List[int]]) -> None:
        row, col = coords
        self.row = row
        self.col = col
        self.free = board[row][col] == 0

    def __eq__(self, __value: object) -> bool:
        if not isinstance(__value, Cell): return False
        return self.row == __value.row and self.col == __value.col

class Node:
    def __init__(self, pos: Cell, cost: int, state: List[Cell], last_move: Optional[int]) -> None:
        self.pos = pos
        self.cost = cost
        self.state = state.copy()
        self.last_move = last_move

    # Return: Depth i.e. nb of parent from root up to `self`
    def depth(self) -> int: return len(self.state)

def up(row: int, col: int): return row - 1, col
def right(row: int, col: int): return row, col + 1
def down(row: int, col: int): return row + 1, col
def left(row: int, col: int): return row, col - 1

UP, RIGHT, DOWN, LEFT = 0, 1, 2, 3
MOVES: Dict = {UP: up, RIGHT: right, DOWN: down, LEFT: left}


# Return: opposite of `move` i.e. LEFT ⇒ RIGHT...
def opposite(move: int): return move + 2 % len(MOVES)

def valid_moves(node: Node, N: int, M: int) -> Set[int]:
    """NxM: dimension of the board. i.e.  N:nb of rows, M: nb of columns.
    Return moves that would not cause an index out of bounds exception.
    (not guaranteed to be actually usable ⇒ doesnt check if next cell is free or not)"""
    out: Set[int] = set(range(UP, LEFT + 1))
    i, j = node.pos.row, node.pos.col
    if i ≥ N-1: out.discard(DOWN)
    if j ≥ M-1: out.discard(RIGHT)
    if i ≤ 0: out.discard(UP)
    if j ≤ 0: out.discard(LEFT)
    return out
```

```python
def listOfChildren(node: Node, board: List[List[int]]) -> Set[int]:
    not_out_of_bounds: Set[int] = valid_moves(node, len(board), len(board[0]))
    # remove last move to avoid having 2 consecutive moves cancelling each other
    if node.last_move is not None: not_out_of_bounds.discard(opposite(node.last_move))
    to_remove: Set[int] = set()
    cell = node.pos
    for move in not_out_of_bounds:
        new_row, new_col = MOVES[move](cell.row, cell.col)
        if board[new_row][new_col] == 1: to_remove.add(move) # if cell is blocked remove it

    return not_out_of_bounds.difference(to_remove)

def cost(cell: Cell, depth: int, goal: Cell) -> int:
    crt_row, crt_col = cell.row, cell.col
    goal_row, goal_col = goal.row, goal.col
    l1_norm = abs(crt_row - goal_row) + abs(crt_col - goal_col) # g(x) i.e. distance
    return depth + l1_norm

def addToLiveNodes(pq: List[Node], node: Node):
    """Emulate behavior of add PriorityQueue """
    pq.append(node)
    for i in range(len(pq) - 1, 0, -1): # iterate from the end
        crt, next = pq[i], pq[i - 1]
        if next.cost < crt.cost :
            pq[i], pq[i - 1] = next, crt # keep sorted decreasing order
        else: break # rest is sorted

def nextENode(pq: List[Node]) -> Node: return pq.pop()

def P(node: Node, goal: Cell): return node.pos == goal


def shortest_path(board: List[List[int]], _start: Tuple[int, int], _goal: Tuple[int, int]) -> Node:
    start, goal = Cell(_start, board), Cell(_goal, board)
    live_nodes = []
    root = Node(start, 0, [], None)
    enode = root
    while not P(enode, goal):
        for move in listOfChildren(enode, board):
            new_cell: Cell = Cell(MOVES[move](enode.pos.row, enode.pos.col), board)
            new_node = Node(new_cell, cost(new_cell, enode.depth() + 1, goal), [*enode.state, enode.pos], move)
            addToLiveNodes(live_nodes, new_node)

        enode = nextENode(live_nodes)
    return enode
```

========== Task scheduling ==========

Dans ce problème, on veut assigner n tâches à n agents, sachant que chaque agent est rémunéré et chaque tâche est assignée à un agent différent.

C'est un problème que l'on peut couramment rencontrer: assigner des emplacements à des bâtiments devant être construits, assigner des évènements à des organisteurs candidats, etc... La rémunération des agents peut être représentée par une matrice des coûts, où l'élément (i, j) représente le coût de l'exécution de la tâche i par l'agent j.

Le but de l'exercice est d'implémenter une stratégie Branch-and-Bound pour résoudre ce problème, avec la matrice des coûts comme paramètre. Pour ce faire, nous proposons de décrire un état, ou solution partielle, par la liste des tâches déjà assignées et des agents déjà assignés (on commence donc avec des listes vides). De cette manière, on peut définir le coût d'une solution partielle par la somme des coûts minimaux pour chaque tâche restante, augmentée du coût effectif des assignations effectuées jusque là. Prenons par exemple la matrice des coûts ci-dessous:

```python
from math import inf

COST_MATRIX = []
class Node:
    def __init__(self, cost: int, state: List[Tuple[int, int]]):
        """affectation: task_index, agent_index , state: list of tuple[task_index, agent_index]"""
        self.cost = cost
        self.state = state.copy()
    def affectation(self) -> Tuple[int, int]: return self.state[-1]

def cost(affectations: List[Tuple[int, int]], cost_mat: List[List[int]]) -> int:
    h = sum(cost_mat[task_idx][cost_idx] for (task_idx, cost_idx) in affectations)  # cost accumulated up to here
    used_idx = {affect[0] for affect in affectations}
    g, k = 0, len(affectations)
    for row in cost_mat[k:]:
        min_idx, min_cost = None, inf  # no minimum yet
        for a_idx, a_cost in enumerate(row):
            if a_cost < min_cost and a_idx not in used_idx:
                used_idx.discard(min_idx)  # agent `min_idx` isn't affected anymore
                min_idx, min_cost = a_idx, a_cost
                used_idx.add(a_idx)
        g += min_cost
    return g + h


def listOfChildren(parent: Node, cost_mat: List[List[int]]):
    old_affectations = parent.state
    used_agents = {affectation[1] for affectation in old_affectations}
    # agent already used in previous tasks
    N, task_index = len(cost_mat), parent.affectation()[0] + 1  # nb of agents & tasks
    free_agents = [idx for idx in range(N) if idx not in used_agents]
    return [
        Node(
            cost=cost(old_affectations + [(task_index, agent_idx)], cost_mat),
            state=old_affectations + [(task_index, agent_idx)],
        )
        for agent_idx in free_agents
    ]

def addToLiveNodes(node: Node, pq: List[Node]):
    """Add while maintining priority queue order"""
    pq.append(node)
    for i in range(len(pq) - 1, 0, -1):
        crt, next = pq[i], pq[i - 1]
        # smallest at the end
        if crt.cost > next.cost: pq[i], pq[i - 1] = next, crt
        else: break  # already sorted

def P(node: Node, cost_mat: List[List[int]]) -> bool:
    return len(node.state) == len(cost_mat)  # affectations completed?


def branch_bound(cost_mat: List[List[int]]) -> Tuple[List[Tuple[int, int]], int]:
    """Return: affectations, i.e. list of coordinates"""
    root_affect = 0, cost_mat[0].index(min(cost_mat[0]))
    root = Node(cost([root_affect], cost_mat), [root_affect])
    live_nodes = []
    enode = root
    while not P(enode, cost_mat):
        for node in listOfChildren(enode, cost_mat):
            addToLiveNodes(node, live_nodes)
        enode = live_nodes.pop()
        print(enode)
    return enode.state, enode.cost
```

=== Other interesting cost function ===

```python
def W(node_val: int):
    # w(i) => weight of node i. indexing starts at 2 (because node values starts at 1 and node 1 has no cost)
    if node_val - 2 ≥ len(WEIGHTS): print(node_val)
    return WEIGHTS[node_val - 2]

def cost(child_val: int, parent: Node):
    # computed as cost up to there (parent) + distance (child's weight) from Minimum weight
    return parent.cost + (W(child_val) - MIN_W)
```

========== 15 Puzzle (taquin) ==========

```python
class M:
    UP, RIGHT, DOWN, LEFT = 0, 1, 2, 3
    ALL = [UP, RIGHT, DOWN, LEFT]

def swap(board: List[List[int | None]], pos1: Tuple[int, int], pos2: Tuple[int, int]):
    r1, c1 = pos1
    r2, c2 = pos2
    tmp = board[r1][c1]
    board[r1][c1] = board[r2][c2]
    board[r2][c2] = tmp

def apply_move(board: List[List[int | None]], move: int, blank_pos: Tuple[int, int]):
    """Apply given move on board (switch relevant cells)
    and returns the new position of the whitespace"""
    mr, mc = 0, 0  # modifiers: row, column
    match move:
        case M.UP: mr -= 1
        case M.RIGHT: mc += 1
        case M.DOWN: mr += 1
        case M.LEFT: mc -= 1
    orow, ocol = blank_pos # old row old col
    nr, nc = orow + mr, ocol + mc
    swap(board, blank_pos, (nr, nc))
    return nr, nc

def count_misplaced(board: List[List[int | None]]) -> int:
    N, misp = len(board), 0
    for i, row in enumerate(board):
        for j, val in enumerate(row):
            if val is not None and i * N + j + 1 ≠ val: misp += 1
    return misp

def cost(board: List[List[int | None]], state: List[int]) -> int:
    """state: List of moves to apply to get the board from which to compute the cost
    start_pos: position of blank space for root node."""
    h = len(state)
    return h + count_misplaced(board)
```

```python
class Node:
    def __init__(self, state: List[int], board: List[List[int | None]], old_bpos: Tuple[int, int]):
        """
        state: list of moves (a move is an int in [0, 3]) to apply to attain
        this node in the game tree.
        NB: state[-1] is the last move made that "connects" to this node
        board: state of the board for parent/adjacent node
        old_bpos: position of the blank space in the parent config
        bpos: position of blank space in this config,
        bd: resulting board after having applied all moves in state"""
        bd = deepcopy(board)
        self.state = state[:]  # copy
        self.bpos = apply_move(bd, state[-1], old_bpos) if state else old_bpos
        self.bd = bd  # storing them all is at worst O(3^(n^2)) space
        self.cost = cost(self.bd, self.state)

    def last_move(self): return self.state[-1] if self.state else None
    def __repr__(self): return f"({moves_to_str(self.state)}, bpos={self.bpos}, cost={self.cost})"

def opposite(move: int) -> int:
    """Return: opposite of `move` i.e. the one that reverse it (UP ⟹ DOWN ...)"""
    return (move + 2) % len(M.ALL)


def next_move_not_out(node: Node) -> Set[int]:
    """Return: sets of next moves that won't cause index out of bounds"""
    N = len(node.bd)
    moves = set(M.ALL)
    row, col = node.bpos
    if row ≤ 0: moves.discard(M.UP)
    if col ≤ 0: moves.discard(M.LEFT)
    if row ≥ N-1: moves.discard(M.DOWN)
    if col ≥ N-1: moves.discard(M.RIGHT)
    return moves

def listOfChildren(node: Node) -> List[Node]:
    next_moves: Set[int] = next_move_not_out(node)
    last_move: Optional[int] = node.last_move()
    if last_move: next_moves.discard(opposite(last_move))
    return [Node(node.state + [move], node.bd, node.bpos) for move in next_moves]

def addToLiveNodes(pq: List[Node], node: Node):
    pq.append(node)
    for i in range(len(pq) - 1, 0, -1):
        crt, next = pq[i], pq[i-1]
        if crt.cost > next.cost:
            pq[i], pq[i-1] = next, crt
        else: break

def P(node: Node) -> bool: return count_misplaced(node.bd) ≤ 0

def fifteen_puzzle(board: List[List[int | None]]) -> Node:
    blank_pos = 0, 0
    for i, row in enumerate(board):
        for j, val in enumerate(row):
            if val is None : blank_pos = i, j

    live_nodes: List[Node] = []
    root = Node([], board, blank_pos)
    enode: Node = root
    while not P(enode):
        for node in listOfChildren(enode): addToLiveNodes(live_nodes, node)
        print([node.cost for node in live_nodes])
        enode = live_nodes.pop()
        print(enode); mprint(enode.bd); print(" ")  # noqa: E702
    return enode
```