Contents

1 Basic Test Results

```
Starting tests...
Mon 01 Jul 2024 21:39:30 IDT
9456ce48838b4aeb94e705d695efe70ce25c0ae2  -


Archive:  /tmp/bodek.51vlmx5k/intro2cs2/ex8/daniel.rez/presubmission/submission
  inflating: src/puzzle_solver.py


Running presubmit code tests...
6 passed tests out of 6 in test set named 'presubmit'.
result_code    presubmit    6    1
Done running presubmit code tests

Finished running the presubmit tests

Additional notes:

The presubmit tests check only for the existence of the correct function names.
Make sure to thoroughly test your code.

```

2 puzzle solver.py

```python
#################################################################
# FILE : image_editor.py
# WRITER : daniel_riazanov , daniel.rez , 336119300
# EXERCISE : intro2cs ex8 2024
# DESCRIPTION: Mastering Backtracking
# STUDENTS I DISCUSSED THE EXERCISE WITH: None
# WEB PAGES I USED: None
# NOTES: None
#################################################################


from typing import List, Tuple, Set, Optional, Union

# We define the types of a partial picture and a constraint (for type checking).
Picture = List[List[int]]
Constraint = Tuple[int, int, int]


def translate_picture(picture: List[List[int]], treat_as: int) -> List[List[int]]:
    """
    Translates the picture by treating all unknown cells (-1) as a given value.

    :param picture: A two-dimensional list representing the picture where -1 indicates unknown cells.
    :param treat_as: The value to treat unknown cells (-1) as (e.g., 0 for black or 1 for white).
    :return: A new picture where all unknown cells are replaced with the treat_as value.

    Function allows flexibility in handling unknown cells by converting them to a specified value.
    Useful for evaluating the maximum and minimum number of seen cells in a modular way.
    """
    # Initializing return value
    translated = []
    # Traverse through matrix and assign to all -1 (unknown cells) the treat_as value
    for row in picture:
        translated.append([treat_as if cell == -1 else cell for cell in row])
    return translated


def count_seen_cells(picture: List[List[int]], row: int, col: int) -> int:
    """
    Counts the number of cells visible from the given cell in the picture, including the cell itself.

    :param picture: A two-dimensional list representing the picture.
    :param row: The row index of the cell.
    :param col: The column index of the cell.
    :return: The number of cells visible from the cell at (row, col).

    Function counts the visible cells in four directions (left, right, up, down) until a black cell (0) is encountered.
    General approach helps determine the visibility of a cell in both max_seen_cells and min_seen_cells functions.
    """
    # No seen cells from black cell
    if picture[row][col] == 0:
        return 0

    n = len(picture)
    m = len(picture[0])
    count = 1

    # Count left (start from same column left cell and go -1 (left) until ind 0)
    for i in range(col - 1, -1, -1):
```

```python
            if picture[row][i] == 0:
                break
            count += 1

        # Count right (start from same column same cell and go +1 (right) until ind border )
        for i in range(col + 1, m):
            if picture[row][i] == 0:
                break
            count += 1

        # Count up (start from same row cell up and go -1 (up) until ind 0 )
        for i in range(row - 1, -1, -1):
            if picture[i][col] == 0:
                break
            count += 1

        # Count down (starts from same row cell down and go +1 (down) until ind border )
        for i in range(row + 1, n):  # Move down
            if picture[i][col] == 0:
                break
            count += 1

        return count


def max_seen_cells(picture: Picture, row: int, col: int) -> int:
    """
    Calculates the maximum number of cells visible from the given cell, treating all unknown cells as white (1).

    :param picture: A two-dimensional list representing the picture.
    :param row: The row index of the cell.
    :param col: The column index of the cell.
    :return: The maximum number of cells visible from the cell at (row, col).

    Function leverages translate_picture to treat all unknown cells as white,
    then uses general count_seen_cells to count the maximum visible cells.
    """
    translated_picture = translate_picture(picture, 1)
    return count_seen_cells(translated_picture, row, col)


def min_seen_cells(picture: Picture, row: int, col: int) -> int:
    """
    Calculates the minimum number of cells visible from the given cell, treating all unknown cells as black (0).

    :param picture: A two-dimensional list representing the picture.
    :param row: The row index of the cell.
    :param col: The column index of the cell.
    :return: The minimum number of cells visible from the cell at (row, col).

    Function leverages translate_picture to treat all unknown cells as black,
    then uses general count_seen_cells to count the minimum visible cells.
    """
    translated_picture = translate_picture(picture, 0)
    return count_seen_cells(translated_picture, row, col)


def is_valid(picture: Picture, constraints_set: Set[Constraint]) -> bool:
    """
    Checks if the given picture satisfies all constraints in the constraints set.

    :param picture: A two-dimensional list representing the picture.
    :param constraints_set: A set of constraints where each constraint is a tuple (row, col, seen).
    :return: True if the picture satisfies all constraints, False otherwise.

    Function ensures that each constraint is satisfied by comparing the number of seen cells with the maximum and
    minimum possible seen cells for each constraint. Built as modular block and is used widely in future functions.
    """
```

```python
    for row, col, seen in constraints_set:
        max_seen = max_seen_cells(picture, row, col)
        min_seen = min_seen_cells(picture, row, col)

        if seen < min_seen or seen > max_seen:
            return False

    return True


def check_constraints(picture: Picture, constraints_set: Set[Constraint]) -> int:
    """
    Checks if the picture satisfies all constraints exactly, partially, or not at all.

    :param picture: A two-dimensional list representing the picture.
    :param constraints_set: A set of constraints  where each constraint is a tuple (row, col, seen).
    :return:  0 if at least 1 constraint is violated, 1 if all constraints are precisely satisfied, 2 otherwise

    Function iterates through each constraint, checks if it is valid, and determines if the constraints are
    exactly or partially satisfied.
    """
    # Initializes a flag to track if all constraints are satisfied exactly
    all_exact = True

    # Iterates through each constraint in the constraints_set
    for row, col, seen in constraints_set:
        # By is_valid function reuse we check If any constraint is not valid and returns 0 (violation).
        if not is_valid(picture, {(row, col, seen)}):
            return 0
        # Calculates the maximum and minimum number of seen cells for the current constraint
        max_seen = max_seen_cells(picture, row, col)
        min_seen = min_seen_cells(picture, row, col)

        # Checks if the current constraint is satisfied exactly
        if seen != max_seen or seen != min_seen:
            all_exact = False

    # Final flag value defines if all are exactly satisfied (1), otherwise 2
    return 1 if all_exact else 2


def backtrack(picture: List[List[int]], constraints_set: Set[Tuple[int, int, int]], n: int, m: int,
              count_solutions: bool) -> Union[Optional[List[List[int]]], int]:
    """
    Backtracking function to find solutions to the puzzle.

    :param picture: A two-dimensional list representing the picture.
    :param constraints_set: A set of constraints where each constraint is a tuple (row, col, seen).
    :param n: The number of rows in the picture.
    :param m: The number of columns in the picture.
    :param count_solutions: A flag indicating whether to count the number of solutions or return one solution.
    :return: The picture if a solution is found (count_solutions False), or number of solutions (count_solutions True).

    Modular function uses a nested recursive function _backtrack to explore possible configurations of the picture
    (with more args_) and ensures that each configuration satisfies each constraint.

    """
    def _backtrack(row: int, col: int) -> Union[Optional[List[List[int]]], int]:
        """
        Nested recursive function to perform the actual backtracking.

        :param row: Current row index.
        :param col: Current column index.
        :return: The picture if solution is found (count_solutions False), or number of solutions (count_solutions True)
        """
        # Base case: if we have reached the end of the last row, checks constraints
        if row == n:
            # Checks if the current configuration satisfies all constraints exactly
```

```python
            if check_constraints(picture, constraints_set) == 1:
                return 1 if count_solutions else picture
            return 0

        # Determines the next cell to process
        next_row, next_col = (row, col + 1) if col + 1 < m else (row + 1, 0)
        solutions_count = 0

        # We try both possible colors (0 for black, 1 for white) for the current cell (Reduces the num of iterations)
        for color in [0, 1]:
            picture[row][col] = color
            # Checks if the current configuration is valid so far
            if is_valid(picture, {(r, c, s) for r, c, s in constraints_set if r == row or c == col}):
                # Recursively call _backtrack for the next cell
                result = _backtrack(next_row, next_col)

                # If counting solutions, accumulates the result
                if count_solutions:
                    solutions_count += result
                # If looking for a single solution, returns the result if a solution is found
                elif result is not None:
                    return result

            # Resets the current cell (backtrack) if no valid configuration found
            picture[row][col] = -1
        # Returns the total number of solutions if counting, otherwise return None
        return solutions_count if count_solutions else None
    # Starts the recursive backtracking from the first cell
    return _backtrack(0, 0)


def solve_puzzle(constraints_set: Set[Constraint], n: int, m: int) -> Optional[Picture]:
    """
    Solves the puzzle by finding one valid configuration of the picture.

    :param constraints_set: A set of constraints where each constraint is a tuple (row, col, seen).
    :param n: The number of rows in the picture.
    :param m: The number of columns in the picture.
    :return: The solved picture or None if no solution exists.

    Function initializes an empty picture and uses the backtrack function to find one valid solution.
    """

    # Initializes an empty picture where all cells are set to -1 (unknown)
    # (represents the initial state where no cells have been colored yet)
    initial_picture = [[-1 for _ in range(m)] for _ in range(n)]
    # Uses the modular backtrack function to find one valid solution
    # count_solutions set False to return the first valid configuration found
    return backtrack(initial_picture, constraints_set, n, m, count_solutions=False)


def how_many_solutions(constraints_set: Set[Constraint], n: int, m: int) -> int:
    """
    Counts the number of valid solutions for the given puzzle.

    :param constraints_set: A set of constraints where each constraint is a tuple (row, col, seen).
    :param n: The number of rows in the picture.
    :param m: The number of columns in the picture.
    :return: The number of valid solutions.

    Function initializes an empty picture and uses the backtrack function to count all valid solutions.
    """
    initial_picture = [[-1 for _ in range(m)] for _ in range(n)]
    # count_solutions is set to True to count all valid configurations found
    return backtrack(initial_picture, constraints_set, n, m, count_solutions=True)


def generate_puzzle(picture: Picture) -> Set[Constraint]:
```

```python
    """
    Generates a set of constraints from the given picture that ensures it is the unique solution to the puzzle.

    :param picture: A two-dimensional list representing the picture.
    :return: A set of constraints where each constraint is a tuple (row, col, seen).

    Function first generates initial constraints based on the given picture and then prunes unnecessary constraints
    to ensure that the solution is unique and minimal.
    """
    n = len(picture)
    m = len(picture[0])
    constraints_set = set()

    # Generates initial constraints based on the picture
    for row in range(n):
        for col in range(m):
            if picture[row][col] == 1:
                # Calculates the number of cells seen from the current cell
                seen = max_seen_cells(picture, row, col)
                # Adds the constraint (row, col, seen) to the constraints set
                constraints_set.add((row, col, seen))

    # Function to check if the puzzle has a unique solution given a set of constraints
    def is_unique_solution(constraints_set: Set[Constraint]) -> bool:
        # Initializes an empty picture where all cells are set to -1 (unknown)
        initial_picture = [[-1 for _ in range(m)] for _ in range(n)]
        # Uses the backtrack function to count all valid solutions
        # Returns True if there is exactly one solution, False otherwise
        return backtrack(initial_picture, constraints_set, n, m, count_solutions=True) == 1

    # Prunes unnecessary constraints to ensure the solution is minimal
    for row, col, seen in list(constraints_set):
        # Creates a temporary copy of the constraints set
        temp_constraints_set = constraints_set.copy()
        # Removes the current constraint from the temporary set
        temp_constraints_set.remove((row, col, seen))
        # We check if the puzzle still has a unique solution without the current constraint
        if is_unique_solution(temp_constraints_set):
            # If the solution is still unique, we remove the constraint from the original set
            constraints_set.remove((row, col, seen))

    return constraints_set
```