Contents

1 Basic Test Results

```
Starting tests...
Tue Nov 30 11:36:18 IST 2021
97a0f83f46dfdb6e300a64c1d6a31cfd1e635555  -


Archive:  /tmp/bodek.Z_cuBu/intro2cs1/ex8/eyalmutzary/presubmission/submission
  inflating: src/puzzle_solver.py


Running presubmit code tests...
--> BEGIN TEST INFORMATION
Test name: presubmit_generate
Module tested: puzzle_solver
Function call: generate_puzzle([[1]])
Expected return value: {(0, 0, 1)}
More test options: {}
--> END TEST INFORMATION
**********************************************************************
********************     There is a problem:
********************     The test named 'presubmit_generate' failed.
**********************************************************************
Wrong result, input: [[[1]]]:
expected: {(0, 0, 1)}
actual:   None
result_code    presubmit_generate    wrong    1
5 passed tests out of 6 in test set named 'presubmit'.
result_code    presubmit    5    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
from typing import List, Tuple, Set, Optional


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


# ---------- Prolog ----------


def create_default_picture(n: int, m: int) -> Picture:
    picture = []
    for i in range(n):
        picture.append([])
        for _ in range(m):
            picture[i].append(-1)
    return picture


def append_constraints_set(picture: Picture, constraint_set: Set[Constraint]) -> List[List[int]]:
    if constraint_set == set():
        return picture
    for constraint in constraint_set:
        picture[constraint[0]][constraint[1]] = constraint[2]
    return picture


def print_picture(picture: List[List[int]]) -> None:
    for i in range(len(picture)):
        for j in range(len(picture[i])):
            if picture[i][j] == -1:
                print(" ? " , end="")
            # elif picture[i][j] == 0:
            #     print("   " , end="")
            # elif picture[i][j] == 1:
            #     print("   " , end="")
            else:
                print(" " + str(picture[i][j]) + " " , end="")
        print()


# ---------- Part 1 ----------


def _should_break(num: int, is_max: bool) -> bool:
    if is_max and num == 0:
        return True
    elif not is_max and num <= 0:
        return True
    return False


def _seen_row(row: List[int], col: int, is_max: bool):
    count: int = 0
    for i in range(col, len(row), 1):
        if _should_break(row[i], is_max):
            break
        count += 1
    for i in range(col-1, -1, -1):
        if _should_break(row[i], is_max):
            break
        count += 1
    if row[col] != 0:
        count -= 1
    return count


def _seen_col(picture: Picture, row: int, col: int, is_max: bool):
    count: int = 0
    for i in range(row, len(picture), 1):
        if _should_break(picture[i][col], is_max):
            break
        count += 1
    for i in range(row, -1, -1):
        if _should_break(picture[i][col], is_max):
            break
        count += 1

    if picture[row][col] != 0:
        count -= 1
    return count


# tested
def max_seen_cells(picture: Picture, row: int, col: int) -> int:
    if picture[row][col] == 0:
        return 0
    else:
        return _seen_row(picture[row], col, True) +\
               _seen_col(picture, row, col, True)


# tested
def min_seen_cells(picture: Picture, row: int, col: int) -> int:
    if picture[row][col] <= 0:
        return 0
    else:
        return _seen_row(picture[row], col, False) + \
            _seen_col(picture, row, col, False)


# tested
def check_constraints(picture: Picture, constraints_set: Set[Constraint]) -> int:
    status = 1
    if constraints_set == set():
        return 1

    for const in constraints_set:
        min_seen = min_seen_cells(picture, const[0], const[1])
        max_seen = max_seen_cells(picture, const[0], const[1])
        if const[2] < min_seen or const[2] > max_seen:
            return 0
        elif const[2] == min_seen and const[2] == max_seen:
            continue
        elif min_seen <= const[2] <= max_seen:
            status = 2
    return status


# def copy_picture(picture: Picture):
#     new_picture = []
#     for i in range(len(picture)):
#         new_picture.append([])
#         for j in range(len(picture[i])):
#             new_picture[i].append(picture[i][j])
#     return new_picture
```

```python


def formal_solution(picture):
    new_picture = []
    for i in range(len(picture)):
        new_picture.append([])
        for j in range(len(picture[i])):
            if picture[i][j] == 0:
                new_picture[i].append(0)
            else:
                new_picture[i].append(1)
    return new_picture


def _solve_puzzle_helper(picture: Picture,
                         ind: int,
                         constraints_set: Set[Constraint],
                         sol: List[Picture]) -> Optional[Picture]:
    check = check_constraints(picture, constraints_set)
    if ind == len(picture) * len(picture[0]):
        if check == 1 and len(sol) == 0:
            sol.append(formal_solution(picture))
        return picture

    row, col = ind // len(picture[0]), ind % len(picture[0])

    if picture[row][col] != -1:
        _solve_puzzle_helper(picture, ind + 1, constraints_set, sol)
        return

    for value in (0, 1):
        if len(sol) == 1 or check == 0:
            return
        picture[row][col] = value
        _solve_puzzle_helper(picture, ind + 1, constraints_set, sol)
    picture[row][col] = -1


def solve_puzzle(constraints_set: Set[Constraint], n: int, m: int) -> Optional[Picture]:
    picture = create_default_picture(n, m)
    append_constraints_set(picture, constraints_set)
    sol = []
    _solve_puzzle_helper(picture, 0, constraints_set, sol)
    if len(sol) > 0:
        return sol[0]
    else:
        return None



def _count_solutions(picture: Picture,
                     ind: int,
                     constraints_set: Set[Constraint],
                     counter: List[int]) -> None:
    check = check_constraints(picture, constraints_set)
    if ind == len(picture) * len(picture[0]):
        if check == 1:
            counter[0] += 1
        return

    row, col = ind // len(picture[0]), ind % len(picture[0])

    if picture[row][col] != -1:
        _count_solutions(picture, ind + 1, constraints_set, counter)
        return

    for value in (0, 1):
        if check == 0:
            return
        picture[row][col] = value
        _count_solutions(picture, ind + 1, constraints_set, counter)
    picture[row][col] = -1



def how_many_solutions(constraints_set: Set[Constraint], n: int, m: int) -> int:
    picture = create_default_picture(n, m)
    append_constraints_set(picture, constraints_set)
    counter = [0]
    _count_solutions(picture, 0, constraints_set, counter)
    return counter[0]



"""
    - switch all 1 to -1
    - loop on the picture:
        - if value is 0 or -2 -> continue
        - else:
            - go for min scan.
            - set the value to the min score
            - set -2 to every square around

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