Solving the Tile Replacement Using CSP:

Github repository: **https://github.com/tmasmaliyev/CSP-Tile_Placement-GWU**

1. Introduction

The problem involves placing tiles of specific shapes on a landscape to hide parts of bushes. The goal is to configure the tiles such that a desired set of bushes remains visible in the landscape. This problem can be modeled as a Constraint Satisfaction Problem (CSP), where the placement of tiles needs to satisfy constraints related to the visibility of the bushes in specific areas of the landscape.

---

2. Problem Description

You are given the following in a text file:

- Landscape: A square grid representing the landscape with bush marked as integers (1, 2, 3, 4 or empty space) indicating the types of bushes growing at each position.
- Tiles: Three types of tiles (Full Block, Outer Boundary, and EL Shape), each of which is 4x4 in size and covers different areas of the grid.
    - **Full Block**: Completely covers a 4x4 area, hiding any bushes underneath.
    - **Outer Boundary**: Covers the outer perimeter of the 4x4 area, leaving the center visible.
    - **EL Shape**: Covers two sides of the 4x4 area, leaving the other parts exposed.
- Target: A specification of how many different types of bushes should remain visible after all tiles are placed.

The tiles must cover the entire landscape, and the challenge is to place them in a manner that satisfies the constraints of visible bushes as per the target.

---

3. Solution Approach

To solve this problem, we modeled it as a Constraint Satisfaction Problem (CSP). The approach utilizes the **Forward-checking algorithm** for constraint propagation and solving the placement of tiles.

---

4. Constraints

The CSP formulation involves the following constraints:

- Tile Placement Constraints:
    - Tiles must be placed such that they do not overlap.
    - The tiles must cover the entire **grid** (i.e., the number of tiles corresponds to the area of the landscape divided by the area of the tile).
- Visibility Constraints:
    - After placing all the tiles, the specific bushes (colors) must remain visible as per the target specification.
- Shape Constraints:
    - The placement of each tile type (Full Block, Outer Boundary, EL Shape) must be respected for the region it covers.

---

5. Algorithm: Forward Checking to prune search space

Forward checking is a technique used in CSPs to prune the search space by ensuring that each tile placement is consistent with the remaining constraints before proceeding further.

At each step, we place a tile in a specific location in the landscape and check whether the current configuration satisfies the constraints (i.e., the number of occurrences cannot be less than in targets).

If placing a tile violates the constraints, we backtrack and try another placement.

---

6. Implementation Details

- Input Parsing:
    - The landscape grid is read from the input file, where each line corresponds to a row in the grid.
    - The tiles and their counts are also parsed, noting how many of each tile type are available for placement.
    - The target (visible bushes) is specified.

- Tile Placement:
    - Tiles are placed one by one, and the Forward-checking algorithm checks the validity of each placement.
    - For each tile, the algorithm ensures that no constraints are violated, especially the visibility of the bushes.
- Visibility Checking:
    - After each tile placement, the algorithm checks which bushes remain visible and updates the state of the landscape accordingly.
- Output:
    - The final landscape after all tiles is placed, showing the visible bushes.

---

7. Results

The solution successfully placed the tiles in the landscape, ensuring that the bushes specified in the target remained visible. The use of **Forward-checking** ensured that only valid tile placements were considered, significantly reducing the search space and improving the efficiency of finding a solution.

---

8. Performance Analysis

The **time complexity** of the tile placement problem with forward checking and backtracking in the worst case is $\boldsymbol{O(n^4)}$, where n is the size of the grid. This is because, for each tile, there are $O(n^2)$ possible placements, and there are $O(n^2)$ tiles to place, leading to a total of $\boldsymbol{O(n^4)}$possible configurations in the worst case.

---

9. Code implementation

Implementation Details

- The input is read from a file such as:

  python solver.py --filepath=./input /1.txt

- To apply unit test on multiple test cases containing in test files, here is the code

python unit_test.py

In the unit test file, there is a directory which takes input as directory name to look up. It is possible to change it to own directory name.

In the code example:

```
> python solver.py --filepath=./input/1.txt
( 0,  0) : OUTER_BOUNDARY
( 4,  0) : OUTER_BOUNDARY
( 8,  0) : OUTER_BOUNDARY
(12,  0) : OUTER_BOUNDARY
(16,  0) : OUTER_BOUNDARY
( 0,  4) : OUTER_BOUNDARY
( 4,  4) : EL_SHAPE
( 8,  4) : EL_SHAPE
(12,  4) : FULL_BLOCK
(16,  4) : FULL_BLOCK
( 0,  8) : FULL_BLOCK
( 4,  8) : FULL_BLOCK
( 8,  8) : EL_SHAPE
(12,  8) : FULL_BLOCK
(16,  8) : FULL_BLOCK
( 0, 12) : EL_SHAPE
( 4, 12) : FULL_BLOCK
( 8, 12) : FULL_BLOCK
(12, 12) : EL_SHAPE
(16, 12) : FULL_BLOCK
( 0, 16) : FULL_BLOCK
( 4, 16) : EL_SHAPE
( 8, 16) : EL_SHAPE
(12, 16) : FULL_BLOCK
```

In this input, it says when in (x, y) starting position, the corresponding tile is placed.