Table of Contents

[Part 1: Free-Flow 2](#_Toc497209174)

[1.1 State Representation and Constraints 2](#_Toc497209175)

[1.2 Implementation 2](#_Toc497209176)

[1.2.1 CSP Implementation 2](#_Toc497209177)

[1.2.2 Smart Implementations 3](#_Toc497209178)

[1.3 Performance Discussion 3](#_Toc497209179)

[1.4 Results 4](#_Toc497209180)

# Part 1: Free-Flow

## State Representation and Constraints

The grids are the variables, the domain consists of all distinct colors appear in the puzzle. The constraint (normal violation test) used to assign each grid is shown below:

If Non-Source:

For adjacent grids:

If (No matching grid)

//No flowing in and out, must make space open for flowing in and out

Must have at least 2 unassigned adjacent grids

Else If (one matching grid)

//with either a flowing in or out, keep one grid open for out/in

Must have at least 1 unassigned grid

Else If (exactly 2 matching grids)

Check passes

Else

//more than two matching grids is a violation of game rule

Check fails

Else:

//source

For adjacent grids:

If (no adjacent matching):

//no flowing in, must leave space for flowing in

Must have at least 1 unassigned grid

If (1 matching color):

//with flow in, must not have more matching grids

Should not have more than one adjacent matching grid

## Implementation

### CSP Implementation

The program solves the Free-Flow problems using CSP method. Each grid in the puzzle is a struct containing information such as value domain, assigned value and its heuristic. The solver function first chooses the grid to assign according grid heuristic values, then try one value from the domain of that grid with the violation tests (normal violation test, forward checking and arc consistency check). If all tests are passed, a recursion of the solver function is called to assign the next grid. If anyone of the tests failed, the solver function tries the next value in the grid domain. If all values fail to pass the tests, the solver function restores the puzzle to its previous state (including values and domains of variables) and returns fail to the previous layer of recursion (back trace) who will try the next value from the grid it assigns.

### Smart Implementations

**Heuristic:**

The heuristic value for each grid is the number of values remaining in the domain of that grid. This enables us to assign the most constraint (with the smallest domain) grids first, which helps reducing the branches to explore before finding the solution.

**Smart Implementation:**

In the smart implementation, the heuristic mentioned earlier is used. In addition, we deployed forward checking to detect failures early. For each newly filled grid, we iterate through the values in the domain of its neighbors (one at a time), and run violation test for the entire puzzle. If any one of its neighbor fail the violation test for all values in domain, conclude that forward checking has failed and perform back track.

**Smarter Implementation:**

For larger puzzles (greater than 10x10), it requires stronger branch-elimination rule to achieve reasonable runtime. In addition of using heuristic and forward checking, we also deployed arc consistency check in the smarter implementation. When a grid is assigned with new value, the assignment needs to pass three tests before it moves on to the next grid. The first test is the normal violation test. It acts as a coarse-grained filter of assignments, because it is relatively fast. If an assignment fails the normal violation test, we can directly skip this value without running forward checking and arc consistency checking. If an assignment passes the normal violation test, it moves on to the forward checking test. Forward checking is less expensive than arc consistency check, having forward checking and normal violation check before the arc consistency check reduces the number of arc consistency check we need to run. We try to avoid the arc consistency check because it is expensive.

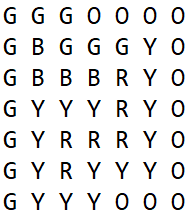
## 1.3 Performance Discussion

The two major methods to improve runtime in this program are to detect failure early and choosing the right variables to assign first. Using the heuristic to assign most constraint variables first dramatically improved the performance. We saw an improvement as much as 60% compare to basic CSP implementation. Forward checking and arc dependency checking can both detect failure early. While Arc dependency check is stronger than forward checking, it is much more expensive to perform, and when the puzzle has sparse assigned grids, it does not eliminate many values. Therefore, arc dependency check only pays off when running during the process of value assignment. As grids get assigned, there are more arc dependencies in the puzzle that significantly improve the ability for arc consistency check to remove values throughout the puzzle. Therefore, we decided to put arc dependency check in our solver function as opposed to only run arc once before trying to solve the puzzle. This significantly helped solving large puzzle(12x12,12x14,14x14) as it brought running time down from 7 hours to 1 to 2 hours. However, due to the high overhead, small puzzles do not benefit from arc dependency check in term of run time. Forward checking is much cheaper than arc consistency check and it is almost constant-time. For this reason, it is beneficial to deploy forward checking for solving puzzles of all sizes.

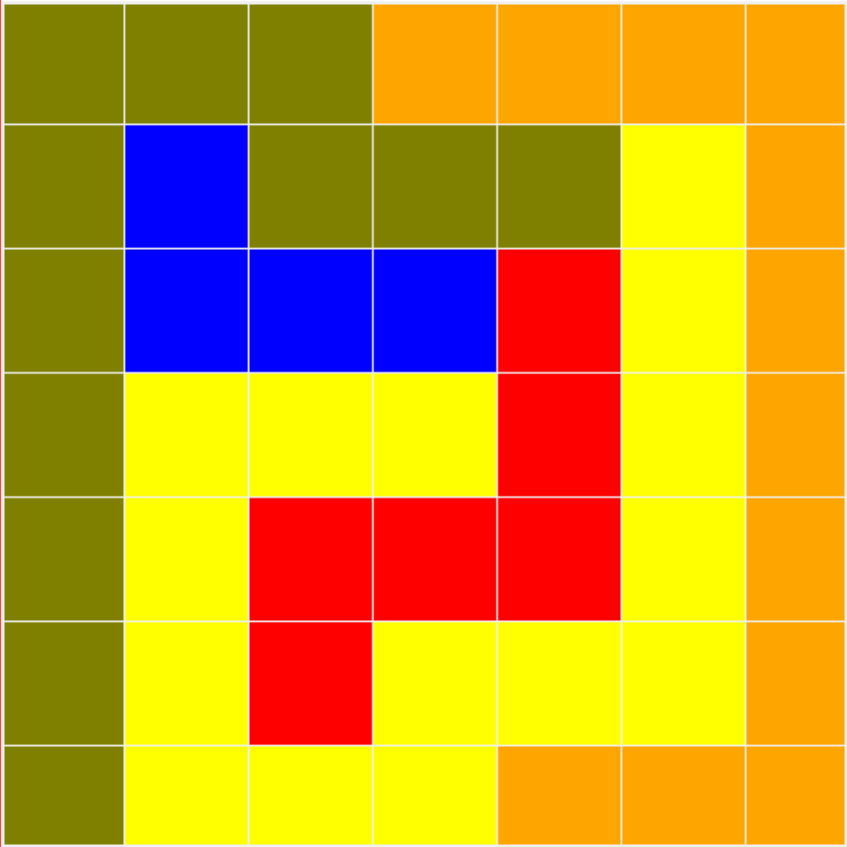
## 1.4 Results

Results (Time in millisecond)

7x7



Visual Representation

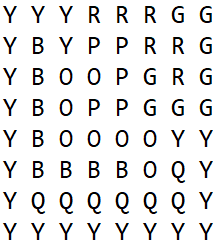


Dumb: Smart

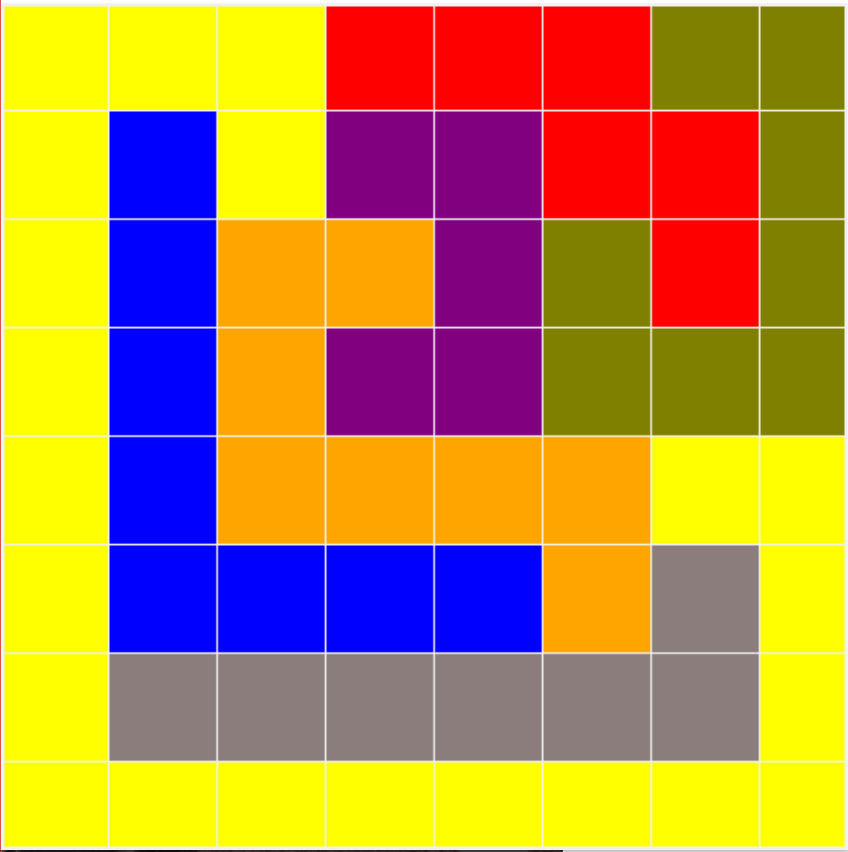
Taking too long numAssignment:816

time: 3194

8x8



Visual Representation

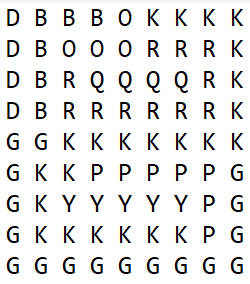


Dumb: Smart

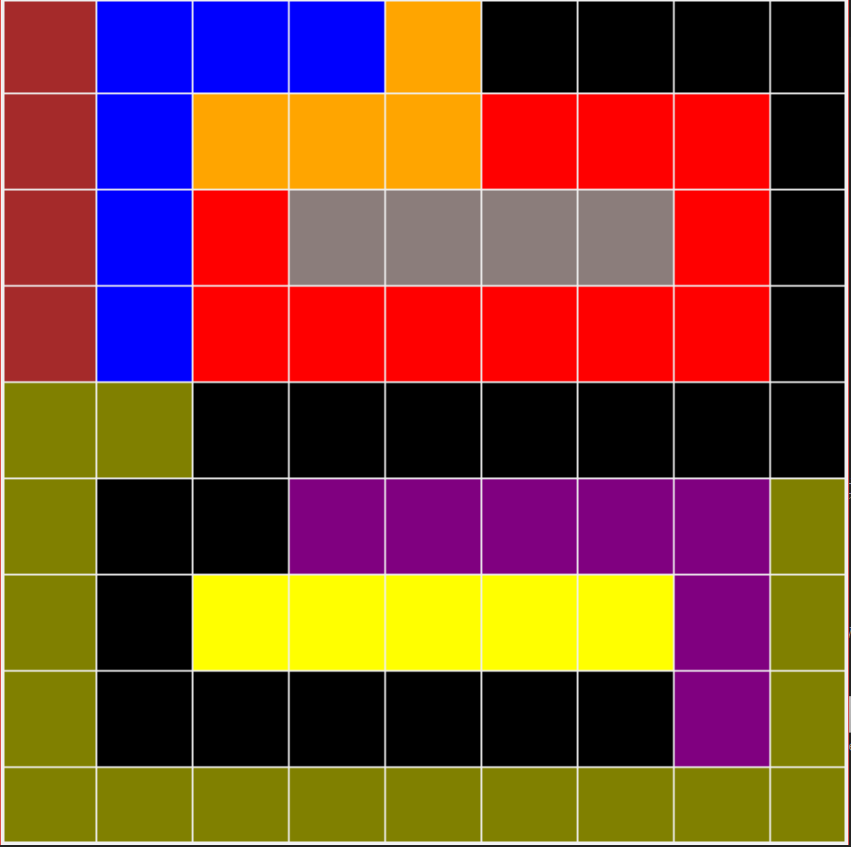
Taking too long numAssignment: 1663

time: 10860

9x9



Visual Representation

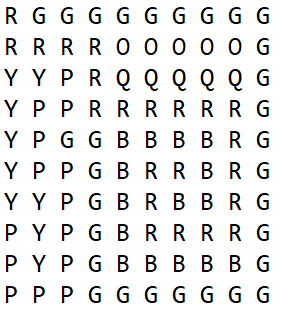


Dumb: Smart

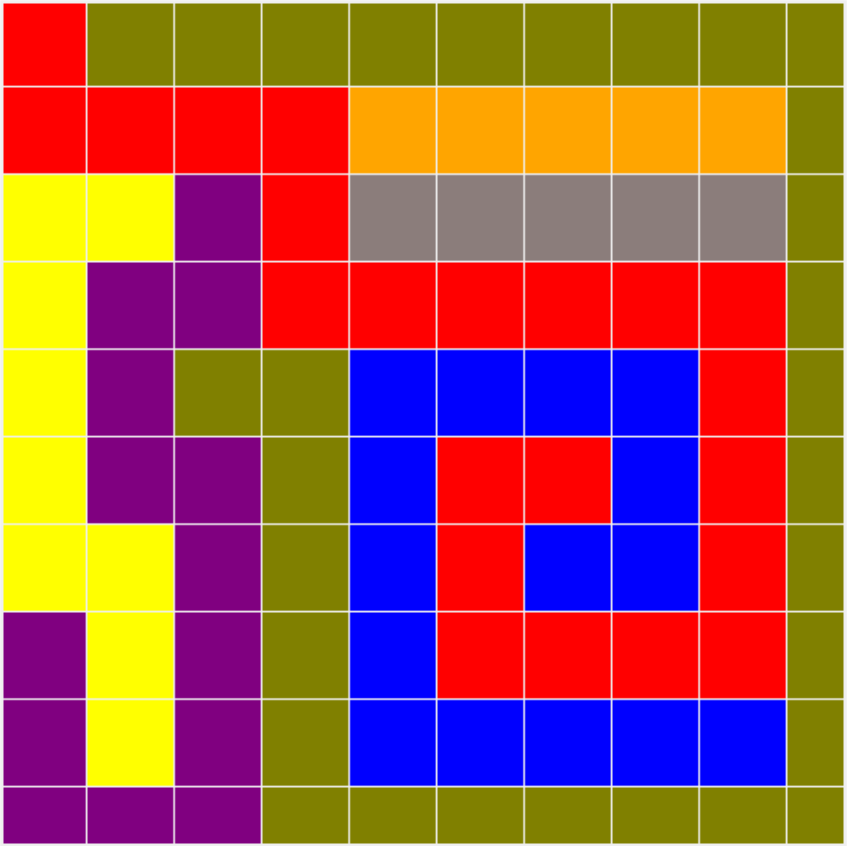
Taking too long numAssignment: 12920

time: 80949

10x10 one



Visual Representation

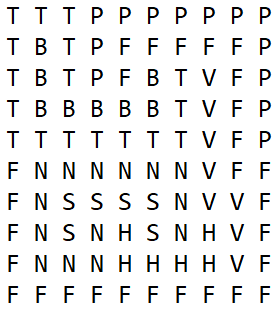


Smart: Smarter

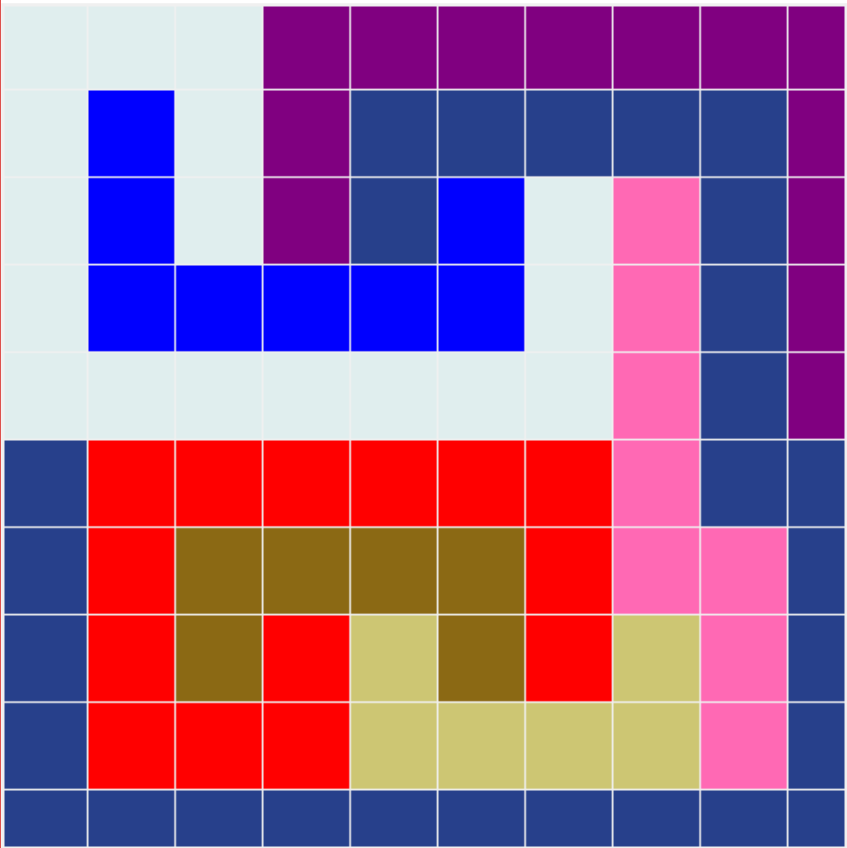
numAssignment:261487 numAssignment:744

time: 22078.7 time: 19489.5

10x10 two



Visual Representation

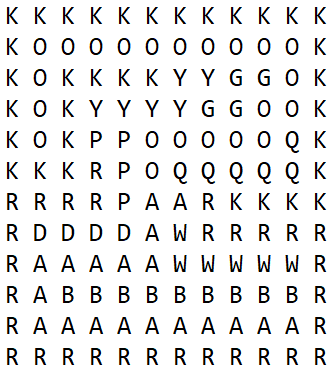


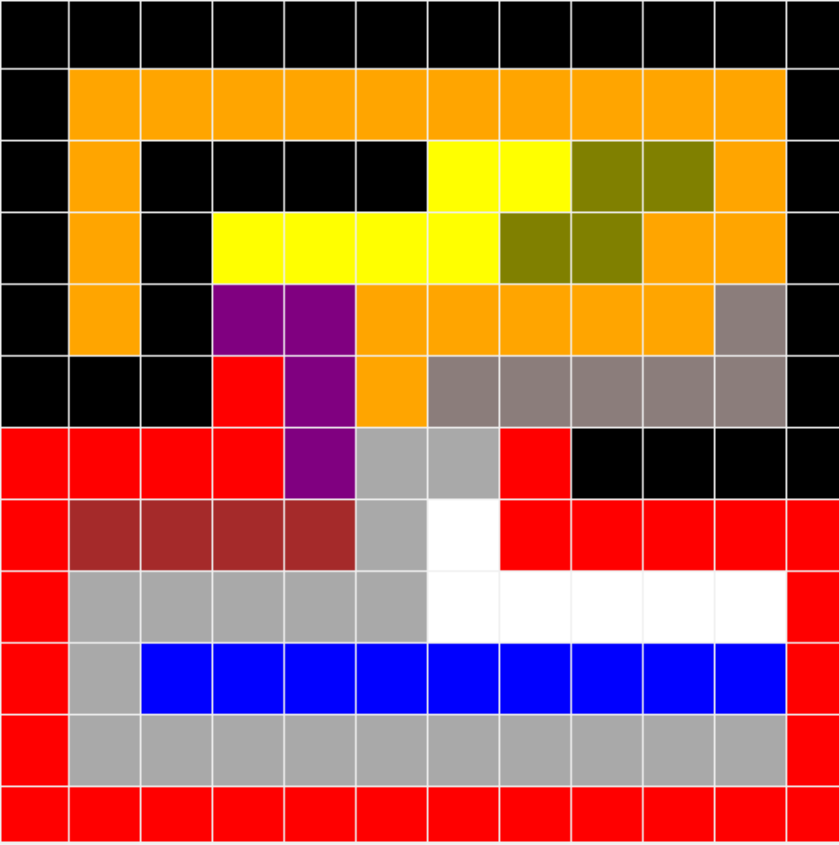
Smart Smarter

numAssignment:207775 numAssignment:413

time: 8336.59 time: 14665

12x12



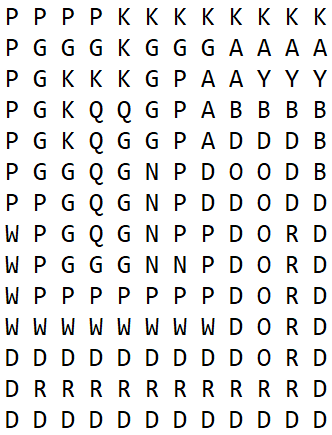


Smarter:

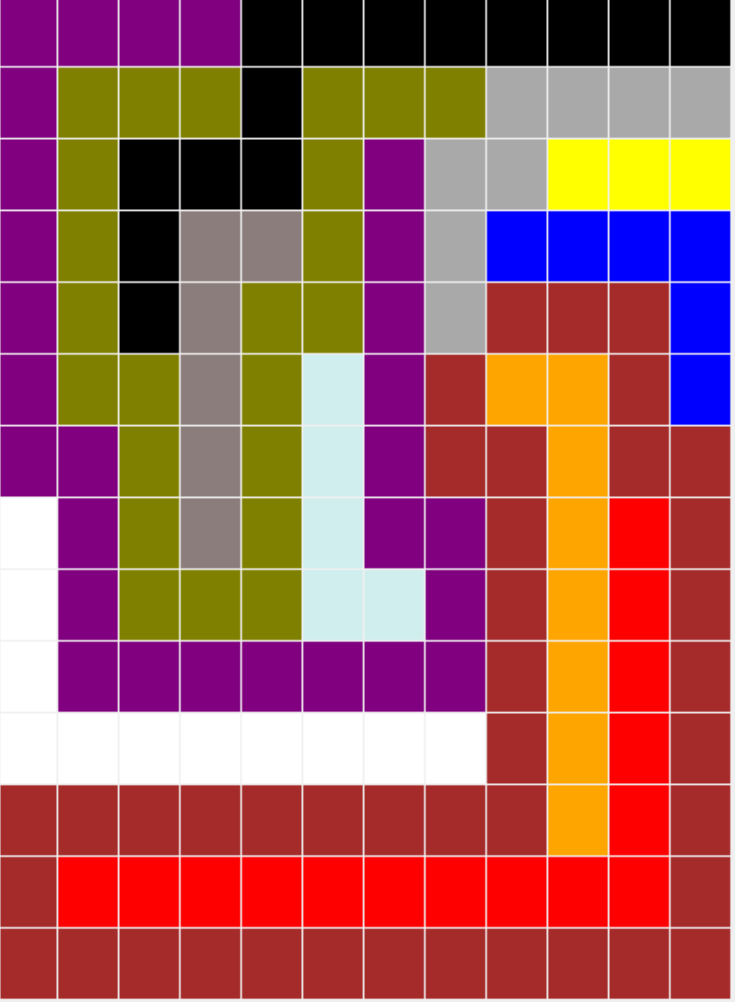
numAssignment:81081

time: 8.55281e+06

12x14



Visual Representation

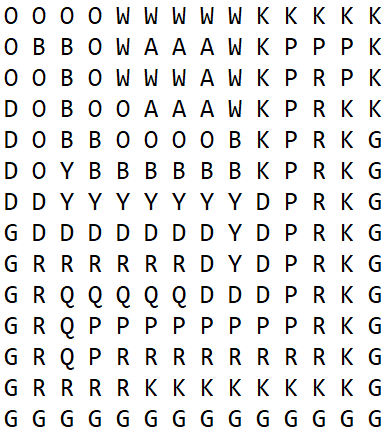


Smarter:

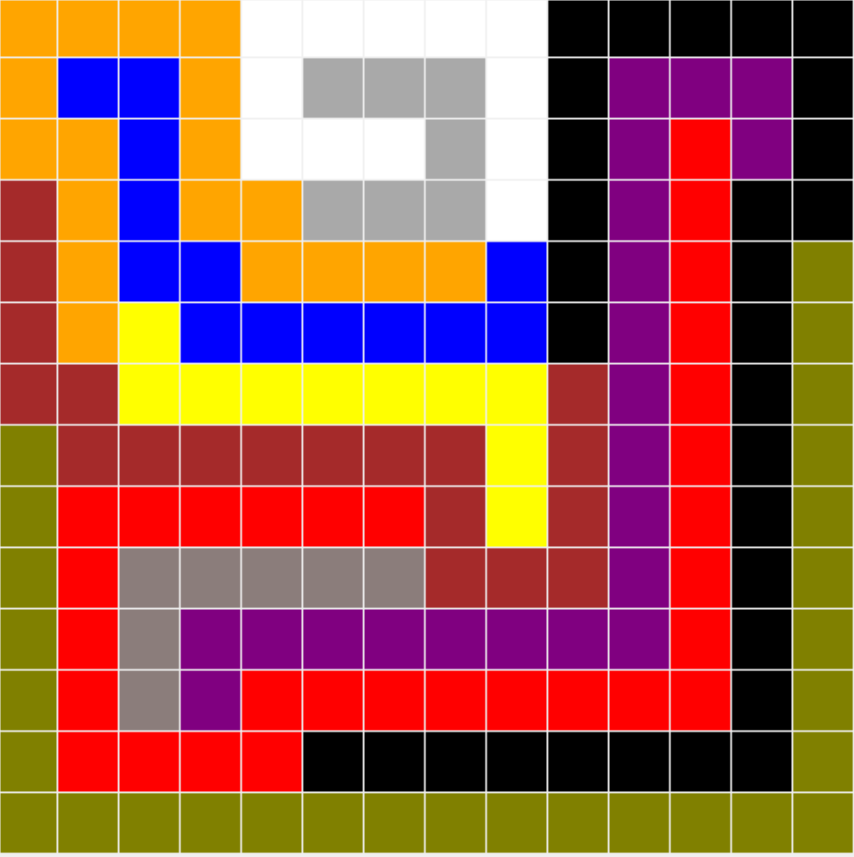
numAssignment:144

time: 19077.8

14x14



Visual Representation



Smarter:

numAssignment:53822

time: 1.07507e+07