Report on the programming project

Authors :
M. ABITBOL Dimitri
M. DE ROUCK Arthur

Supervisors :
M. RAPAPORT Solal
M. LOISEAU Patrick

April 3, 2025

Contents

1 Introduction

This report presents our work (ABITBOL Dimitri and DE ROUCK Arthur) on the ENSAE first-year programming project. The code and the problem description are available on our GitHub repository at the following address:

`https://github.com/arthurdrk/Color-grid-game`

1.1 Project structure

Overall, the project has been organized as a Python package named `color_grid_game`. The main package includes the `Grid` module and the `Solver` module, both of which have been implemented. The subpackage `color_grid_game.solvers` contains the implemented solvers. The initializers allow for direct importation of the classes; typically, the statement

```
from color_grid_game import *
```

imports the `Grid` class, game interface, solvers, etc. It also imports the standard libraries used. The import of subpackages is also implemented to load only the necessary modules.

The code has been generally documented and type-hinted, and sometimes commented when it is less intuitive. Documentation has been generated using the Sphinx module and makes it easier to navigate the project. It is accessible on GitHub Pages at the address:

`https://arthurdrk.github.io/Color-grid-game/`

Additionally, a tests folder is present at the root and contains a set of unit tests. It is based on the test set that was provided and which we have expanded. It contains a total of 68 tests that run in 0.127 seconds on our personal computers.

2 Naive solution

2.1 Greedy Algorithm

Our first approach was a greedy algorithm, implemented in `Solver_Greedy`, which constructs a pairing of grid cells by making locally optimal decisions. It starts by computing all valid neighboring cell pairs according to the given rules using `all_pairs`. Then, it iterates over the grid in row-major order, selecting for each unvisited cell the pair with the minimum cost (computed via `grid.cost`) that includes an unvisited neighbor. This pair is added to the result, and both cells are marked as used, continuing until all cells are processed or no further pairs are possible.

The time complexity is $\mathcal{O}(nm)$, where n and m are the grid's rows and columns, respectively. This arises from processing each of the nm cells once, with a constant-time operation to select the minimum-cost pair from a small set of neighbors. The space complexity is $\mathcal{O}(nm)$, due to storing all pairs and the set of used cells.

2.2 Upgraded Greedy Algorithm

The `Solver_Greedy_Upgraded` improves upon the basic greedy approach by exploring all possible starting points and retaining the pairing with the lowest score. It computes all valid pairs similarly but then iterates over each cell as a starting point. For each, it performs the greedy pairing by traversing the grid in a shifted order (using modulo to wrap around), evaluates the resulting score with `score`, and keeps the pairing with the minimum score across all trials.

The time complexity rises to $\mathcal{O}((nm)^2)$, as the $\mathcal{O}(nm)$ greedy process is repeated nm times for each starting cell. The space complexity remains $\mathcal{O}(nm)$, as storage needs for pairs and used cells do not increase significantly with the iterations.

Below, we compare the performance of both algorithms:

Table 1: Scores

Grid	Greedy	Upgraded
grid00	14	12
grid01	8	8
grid02	1	1
grid03	2	2
grid04	4	4
grid05	39	39
grid11	26	26
grid12	21	21
grid13	22	22
grid14	29	27
grid15	23	21
grid16	32	30
grid17	294	276
grid18	291	267
grid19	284	268
grid21	1850	1846
grid22	1889	1883

Table 2: Times (seconds)

Grid	Greedy	Upgraded
grid00	0.0000	0.0000
grid01	0.0000	0.0000
grid02	0.0000	0.0000
grid03	0.0000	0.0062
grid04	0.0000	0.0009
grid05	0.0000	0.0000
grid11	0.0000	0.0557
grid12	0.0000	0.0551
grid13	0.0000	0.0528
grid14	0.0045	0.0582
grid15	0.0000	0.0585
grid16	0.0000	0.0561
grid17	0.0000	0.0610
grid18	0.0000	0.0533
grid19	0.0003	0.0583
grid21	0.0750	696.1685
grid21	0.0998	574.9029

We achieve slightly better results with `Solver_Greedy_Upgraded` at the cost of much higher computation times for large grids.

3 Maximal Matching approach

In this section, we examine a simplified version of the problem by considering a grid where all values are identical. Without loss of generality, assume that all grid values are equal to 1.

We model the grid as a bipartite graph $G = (U \cup V, E)$, where:

- U includes cells (i, j) with $i + j$ even.
- V includes cells (i, j) with $i + j$ odd.
- Edges $e = (u, v) \in E$ exist between adjacent and color-compatible cells $u \in U$ and $v \in V$.

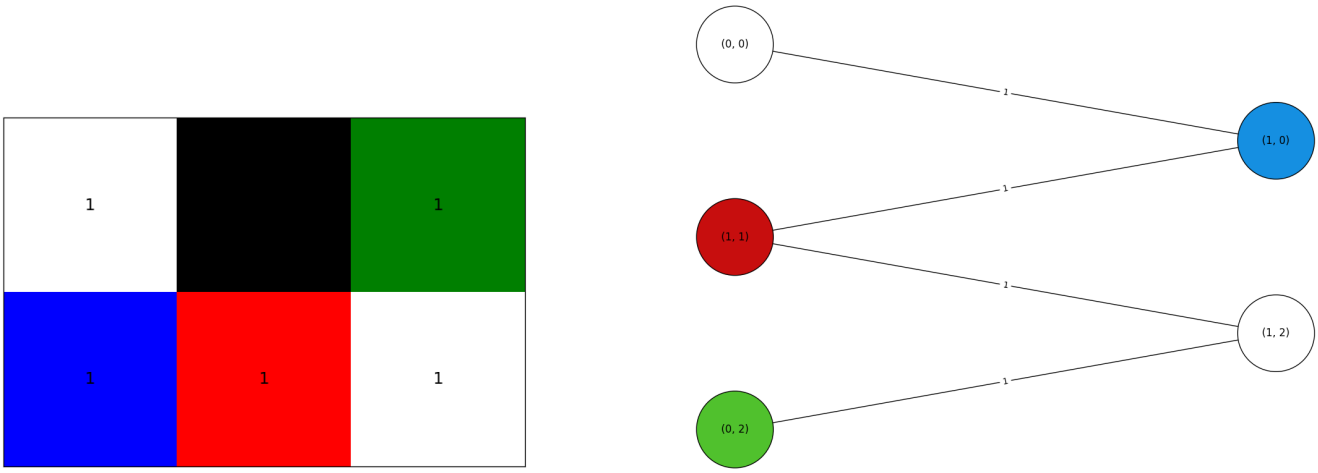

Figure 1: Visualization for `grid01.in`

The formula for the total score S to minimize is given by:

$$S = \sum_{(p_1,p_2)\in \text{pairs}} \text{cost}(p_1, p_2) + \sum_{\substack{(i,j)\notin \text{taken} \\ (i,j) \text{ non-black}}} \text{value}(i, j)$$

In the case where all values are equal to 1, $\text{cost}(p_1, p_2) = 0$ and the score can be simplified as:

$$S = \sum_{\substack{(i,j)\notin \text{taken} \\ (i,j) \text{ non-black}}} \text{value}(i, j) = \sum_{\substack{(i,j)\notin \text{taken} \\ (i,j) \text{ non-black}}} 1 = \text{Card}(\{\text{Unpaired cells}\} \cap \{\text{Non-black cells}\})$$

The goal is thus to find a **maximum matching** in this graph, where each vertex is incident to at most one edge, maximizing the number of matched pairs and minimizing unpaired cells.

3.1 Edmonds-Karp algorithm

In `Solver_Ford_Fulkerson` module, we implemented the Edmonds-Karp algorithm, a specific variant of the Ford-Fulkerson algorithm, to compute the maximum flow in the flow network derived from the bipartite graph of the grid, which corresponds to the maximum matching. This graph is constructed from the original bipartite graph, with an added source vertex `s` connected to each even cell and a sink vertex `t` connected to each odd cell. Below is an example for the previous grid:

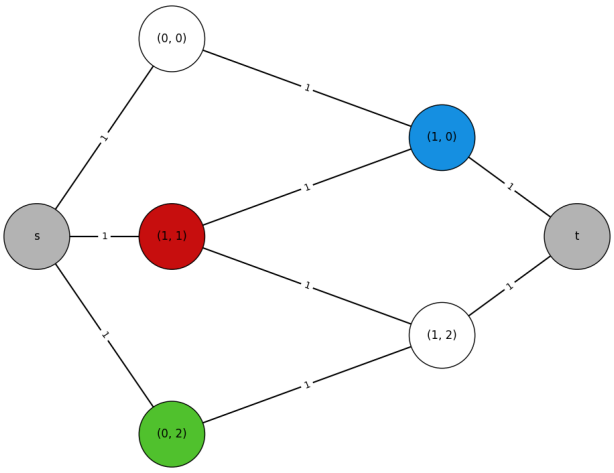

Figure 2: Constructed network for `grid01.in`

Below is some pseudo-code detailing the Edmonds-Karp algorithm we implemented:

Algorithm 1 Maximum Bipartite Matching via Maximum Flow (Edmonds-Karp Variant)

Require: A bipartite graph with even cells U, odd cells V, and valid pairs $E_{bipartite} \subseteq U \times V$
Ensure: A maximum matching M

Step 1: Construct the Flow Network
$G = \Big(U \cup V \cup \{s,t\},\ \{(s,u) : u \in U\} \cup \{(u,v) : (u,v) \in E_{\text{bipartite}}\} \cup \{(v,t) : v \in V\}\Big)$
Step 2: Compute the Maximum Flow via Edmonds-Karp
Initialize the residual graph $G_f \leftarrow G$
while an augmenting path p from s to t exists in G_f (found by BFS) **do** ▷ All edges have unit capacity, so the flow to add is 1.
    for all edges (u, v) in path p **do**
        Remove edge (u, v) from G_f
        Add reverse edge (v, u) to G_f
    end for
end while
Step 3: Extract the Matching
$M \leftarrow \{(u, v) \mid v \in V,\ u \in U,$ and the reverse edge (v, u) exists in $G_f\}$
return M

3.2 Time and Space Complexity

The Edmonds-Karp algorithm's time complexity is $\mathcal{O}(VE^2)$, where V represents the number of vertices and E the number of edges in the flow network. This complexity arises because:

- Each BFS traversal takes $\mathcal{O}(E)$ time.
- The maximum number of augmenting paths is $\mathcal{O}(VE)$.
- Therefore, the total time complexity is $\mathcal{O}(VE^2)$.

For an $n \times m$ grid, $V = \mathcal{O}(nm)$ and $E = \mathcal{O}(nm)$ gives a complexity in $\mathcal{O}(n^3m^3)$. While theoretically high, the algorithm remains practical for moderate grids due to efficient BFS traversal.

The space complexity is $\mathcal{O}(V + E) = \mathcal{O}(nm)$, driven by storing the residual graph's vertices and edges. This linear scaling relative to grid size enables handling large grids without excessive memory demands.

3.3 Results

The solver was tested on various grids, yielding optimal solutions with reasonable computation times:

Grid	Scores	Time (seconds)
grid00.in	0	0.0000
grid01.in	1	0.0000
grid02.in	1	0.0000
grid03.in	2	0.0012
grid04.in	4	0.0000
grid05.in	3	0.0000
grid11.in	26	0.0010
grid12.in	19	0.0009
grid13.in	22	0.0010
grid14.in	27	0.0020
grid15.in	21	0.0020

Grid	Scores	Time (seconds)
grid16.in	28	0.0020
grid17.in	24	0.0010
grid18.in	29	0.0020
grid19.in	25	0.0010
grid21.in	1686	7.2196
grid22.in	1689	6.9112
grid23.in	1711	7.9655
grid24.in	2422	11.1172
grid25.in	2434	11.8905
grid26.in	2359	13.3040

Table 3: Scores and computation times for `Solver_Ford_Fulkerson`.

4 Primal-Dual Hungarian Algorithm Approach

In this section, we examine a more general version of the problem by considering a grid where the values are not necessarily identical. The goal is to find a heuristic for edge weights that reduces our problem to a minimum weight matching problem.

We have implemented the `Solver_Hungarian` module which contains the construction of the heuristic and which solves the minimum weight matching problem using the Hungarian Primal-Dual algorithm.

4.1 Primal Problem

We construct a cost matrix C based on the grid values and the compatibility of neighboring cells. Each entry $C[i, j]$ represents the heuristic cost of pairing cell i (an even cell) with cell j (an odd cell). The cost is computed using a function that considers both the values of the cells and the cost of the pair to avoid local minima.

For example, if cells i and j are compatible and have values v_i and v_j, respectively, the heuristic cost $C[i, j]$ is defined as:

$$C[i, j] = \text{cost}(i, j) - v_i - v_j = |v_i - v_j| - v_i - v_j$$

If a pairing between cells i and j is impossible, we set $C[i, j] = 0$.

In cases where the number of odd cells is not equal to the number of even cells, we fill the cost matrix by considering all possible pairs and assigning a cost of 0 to incompatible pairs. This ensures that the cost matrix is complete and can be used for further computations.

The primal problem seeks to minimize the total assignment cost. Mathematically, it is formulated as follows:

$$
\begin{aligned}
\text{Minimize} \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} C[i,j] \cdot x_{ij} \\
\text{subject to} \quad & \sum_{j=1}^{n} x_{ij} = 1 \quad \forall i = 1, \ldots, n \quad \text{(one column per row)} \\
& \sum_{i=1}^{n} x_{ij} = 1 \quad \forall j = 1, \ldots, n \quad \text{(one row per column)} \\
& x_{ij} \geq 0 \quad \forall i, j
\end{aligned}
$$

Here, x_{ij} is a binary variable: $x_{ij} = 1$ if row i is assigned to column j, and $x_{ij} = 0$ otherwise. The constraints ensure a bijective assignment.

4.2 Dual Problem

The dual problem is constructed from the primal constraints. We associate a dual variable u_i with each row constraint and a dual variable v_j with each column constraint. The dual problem is:

$$
\begin{aligned}
\text{Maximize} \quad & \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \\
\text{subject to} \quad & u_i + v_j \leq C[i,j] \quad \forall i, j
\end{aligned}
$$

Under the framework of linear programming duality, the primal and dual problems are equivalent in the sense that their optimal values coincide (strong duality). A quick proof of this equivalence is provided in Appendix 6.1.

The Hungarian Algorithm iteratively adjusts dual variables u and v (respectively rows and columns potentials) to find optimal assignments via reduced cost minimization. For each row, it searches for augmenting paths by tracking minimal reduced costs, updates dual variables to tighten edges, and backtracks to refine assignments until all rows are optimally matched. The pseudocode for this algorithm is provided in Appendix 6.2.

4.3 Time and Space Complexity

The time complexity of the implementation is $\mathcal{O}(n^3)$. This is because the algorithm involves finding an augmenting path for each row, which can take $\mathcal{O}(n^2)$ time in the worst case, and this process is repeated n times.

The space complexity is $\mathcal{O}(n^2)$ due to the storage of the cost matrix and additional arrays used for tracking paths and assignments. The cost matrix itself requires $\mathcal{O}(n^2)$ space, and the other arrays (like u, v, *path*, *col_to_row*, and *row_to_col*) each require $\mathcal{O}(n)$ space.

4.4 Results

The performance of `Solver_Hungarian` is summarized in the table below. We obtain optimal scores for all grids with a computation time of 1 minute 30 seconds at most for large grids, which is very satisfactory and much better than what we obtained with the naive approach.

Grid	Scores	Time (seconds)
grid00	12	0.0000
grid01	8	0.0000
grid02	1	0.0000
grid03	2	0.0000
grid04	4	0.0000
grid05	35	0.0000
grid11	26	0.0040
grid12	19	0.0070
grid13	22	0.0055
grid14	27	0.0103
grid15	21	0.0098
grid16	28	0.0056

Grid	Scores	Time (seconds)
grid17	256	0.0130
grid18	259	0.0072
grid19	248	0.0087
grid21	1686	52.9826
grid22	1689	50.5171
grid23	1711	49.9223
grid24	2422	85.0853
grid25	2434	86.1241
grid26	2359	82.1262
grid27	23399	90.3288
grid28	23121	95.8540
grid29	23252	92.3099

Table 4: Scores and computation times for `Solver_Hungarian`.

5 Additional features

5.1 New adjacency rule

We have considered a new version of the game where the adjacency constraint for white cells is removed. In this case, the graph corresponding to possible pairings in the grid is generally not bipartite, so the previously implemented solvers do not apply.

The only modification made is the addition of a `rules` parameter in the `all_pairs` method of the `Grid` class, which allows obtaining the list of possible pairs according to the chosen adjacency rule.

We have explored the Edmonds-Blossom algorithm for minimum weight matching, which is applicable to both bipartite and non-bipartite graphs. Although we won't delve into the algorithm's specifics here, as it is not the primary focus of this project, the reader can find detailed information in the reference: *Edmonds' Blossom Algorithm*, Shoemaker, A., Vare, S. (2016). The pseudo-code is also available in Appendix 6.3. However, an implementation of this algorithm, taken from `networkx.max_weight_matching` can be found in the `Solver_Blossom` module, and its performance metrics are summarized in the table below.

Grid	Scores	Time (seconds)
grid00	6	0.0000
grid01	8	0.0000
grid02	1	0.0000
grid03	0	0.0000
grid04	0	0.0000
grid05	21	0.0127
grid11	0	0.0219
grid12	1	0.1053

Grid	Scores	Time (seconds)
grid13	0	0.0666
grid14	11	0.0479
grid15	7	0.0526
grid16	22	0.0465
grid17	136	0.1680
grid18	131	0.1283
grid19	98	0.1326

Table 5: Scores and computation times for `Solver_Blossom`.

5.2 Two players game and bots

We have considered the two-player version of the game, with the following rules: each player in turn chooses a pair of cells, and their score increases by the cost of the pair. The player with the lowest score when there are no pairs left wins the game. In the event of a tie, the player who played fastest wins.

We have implemented two versions of bots that play very differently, and their operation is detailed below.

5.2.1 MiniMax Bot

The `MiniMax_Bot` module utilizes a two-ply lookahead strategy to balance immediate gains with opponent counterplay. It evaluates each possible move by considering the opponent's greedy optimal response. The method `move_to_play` operates as follows:

- **Grid Simulation**: For each candidate pair, a grid copy is created with the selected cells marked as forbidden (black). This simulates the game state after the bot's move.
- **Opponent Modeling**: The bot assumes the opponent will greedily select the lowest-cost remaining pair. The opponent's best move is identified using a min-cost search.
- **Score Calculation**: The bot prioritizes moves that minimize its own cost while maximizing the opponent's subsequent cost. The score combines both factors, favoring pairs where the opponent is forced into high-cost options.

This approach ensures local optimality with a complexity of $\mathcal{O}((nm)^2)$, suitable for moderate grid sizes. The bot sacrifices deeper lookahead for computational efficiency, focusing on immediate turn interactions.

5.2.2 Monte Carlo Tree Search (MCTS) Bot

The `MCTS_Bot` module employs stochastic simulations to model long-term outcomes, utilizing a core process that integrates:

- **Rollout Simulations**: For each initial move, S full-game simulations are performed. During these rollouts, both players alternate using an ε-greedy strategy – choosing the lowest-cost pair with probability $(1 - \varepsilon)$ or a random top-5 pair otherwise.
- **Score Averaging**: The average score differential (bot's score minus opponent's) across all simulations determines the quality of an initial move. The pair with the best average is selected.
- **Exploration-Exploitation Trade-off**: The ε-greedy strategy balances exploitation (optimal immediate moves) and exploration (sampling suboptimal but promising paths), mimicking imperfect human decision-making.

While computationally intensive ($\mathcal{O}(S \cdot n^3 m^3)$), this method outperforms heuristic-based bots in complex scenarios by evaluating diverse game trajectories. The fixed simulation count (S) allows tuning between accuracy and runtime.

5.3 Graphical user interface (GUI)

The ColorGrid game features a Pygame-based GUI launched via `run_game.py`, enabling interactive puzzle-solving with real-time feedback on scores, timers, and turns. The `UIManager` class renders the grid, buttons (restart, solution, menu), and dynamic text, supporting single-player, two-player, or Bot VS modes (Stockfish (`MiniMax`) or DeepBlue (`MCTS`)), along with classic or "No Adjacency" rule variants.

The `Game` class manages the core loop, linking player inputs to solver logic via `SolverManager`. Grid selection is streamlined through the `GridManager`, offering a scrollable, color-coded list of grid files sorted by difficulty. The interface adapts to window resizing and includes an accessible rules screen. While optimized for small-to-medium grids, larger grids may experience slight rendering delays. Some screenshots of the UI are available in Appendix 6.4.

6 Appendix

6.1 Primal-Dual Equivalence Proof

Theorem 1 (Strong Duality for Assignment Problems)**.** *For the primal assignment problem (minimization) and its dual (maximization), the optimal values satisfy:*

$$\sum_{i=1}^{n}\sum_{j=1}^{n} C[i,j]\cdot x_{ij}^{*} = \sum_{i=1}^{n} u_i^{*} + \sum_{j=1}^{n} v_j^{*}$$

where x_{ij}^{*} is the optimal primal solution and (u_i^{*}, v_j^{*}) are the optimal dual variables.

Proof. The assignment problem is a special case of a linear program (LP) with totally unimodular constraint matrices. By the LP duality theorem (admitted here), if the primal has an optimal solution, so does the dual, and their objective values are equal.

1. **Primal Feasibility**: By construction, x_{ij}^{*} satisfies the assignment constraints.
2. **Dual Feasibility**: The dual variables u_i^{*}, v_j^{*} satisfy $u_i^{*} + v_j^{*} \leq C[i,j]$.
3. **Complementary Slackness**: At optimality, for all i, j:

$$x_{ij}^{*} > 0 \implies u_i^{*} + v_j^{*} = C[i,j].$$

Summing over all i, j:

$$\sum_{i,j} C[i,j]x_{ij}^{*} = \sum_{i} u_i^{*}\left(\sum_{j} x_{ij}^{*}\right) + \sum_{j} v_j^{*}\left(\sum_{i} x_{ij}^{*}\right) = \sum_{i} u_i^{*} + \sum_{j} v_j^{*}.$$

Thus, the primal and dual objectives are equal at optimality. □

6.2 Hungarian Algorithm Pseudocode

Algorithm 2 Primal-Dual Hungarian Algorithm Implementation

Require: Cost matrix C of size $n \times n$
Ensure: Optimal assignment pairs (row_ind, col_ind)

Initialize dual variables $u \leftarrow \{0\}^n$, $v \leftarrow \{0\}^n$
Initialize assignments $row\_to\_col \leftarrow \{-1\}^n$, $col\_to\_row \leftarrow \{-1\}^n$
for each $current\_row \in \{0, n-1\}$ **do**
    Initialize $path \leftarrow \{-1\}^n$, $min\_value \leftarrow 0$, $sink \leftarrow -1$
    Mark $current\_row$ as visited
    while no augmenting path found **do**
        for all columns j **do**
            Compute reduced cost $r = C[current\_row][j] - u[current\_row] - v[j]$
            Track minimal costs and update $path$
        end for
        Find column j with minimal reduced cost
        if minimal cost is ∞ **then**
            return Error (No solution exists)
        end if
        Update min_value and dual variables u, v
        if j is unassigned **then**
            $sink \leftarrow j$
        else
            Move to $row\_to\_col[j]$ and continue search
        end if
    end while
    Backtrack $path$ to update assignments
end for
return Optimal pairs from row_to_col

6.3 Edmonds-Blossom Pseudocode

Algorithm 3 Edmonds-Blossom Algorithm

```
procedure EdmondsBlossom(G)
    Initialize matching M ← ∅
    while there exists an augmenting path P in G do
        M ← M ⊕ P                                   ▷ Symmetric difference
    end while
    return M
end procedure
procedure FindAugmentingPath(G, M)
    Initialize forest F ← ∅
    Initialize queue Q
    for each free vertex v in G do
        Add v to Q
        Add v to F as a root
    end for
    while Q is not empty do
        v ← dequeue(Q)
        for each edge (v, u) in G do
            if u is free then
                return augmenting path from v to u
            else if u is in F and (v, u) is not in M then
                Blossom contraction on cycle containing v and u
            else if u is not in F then
                Add u to F
                Enqueue u
            end if
        end for
    end while
    return no augmenting path found
end procedure
procedure BlossomContraction(C)
    Contract cycle C into a single vertex
    Update graph G and matching M accordingly
end procedure
```

6.4 UI Screenshots

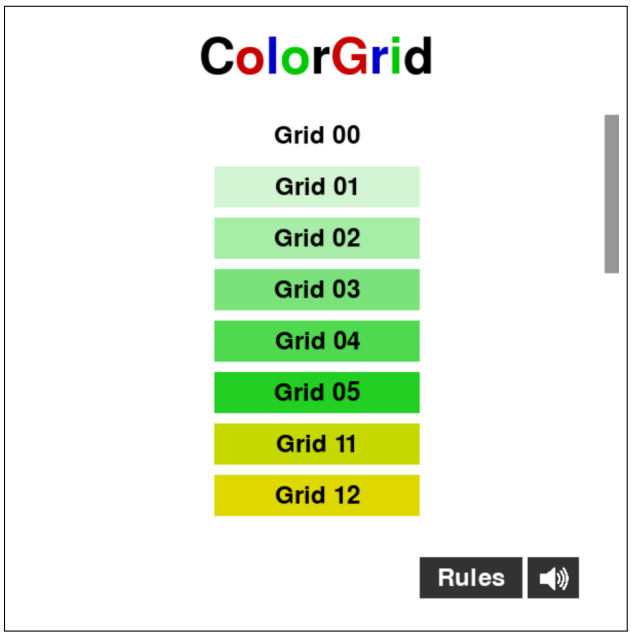

Figure 3: Grid selection

Figure 4: Game mode selection

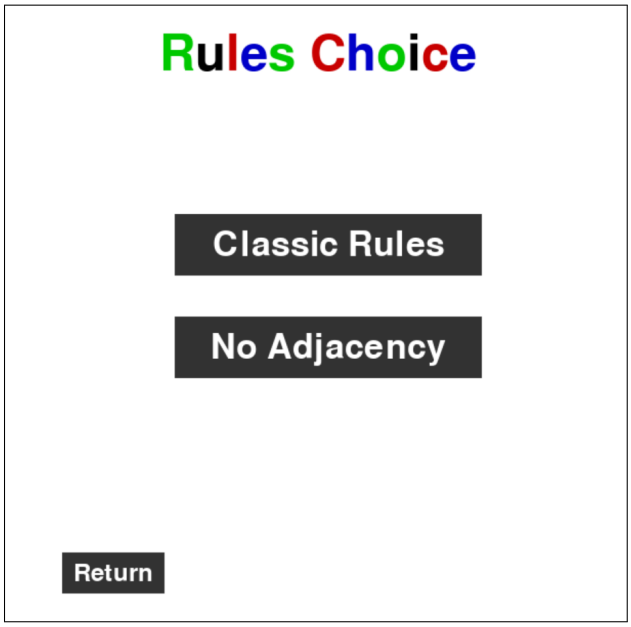

Figure 5: Rules selection

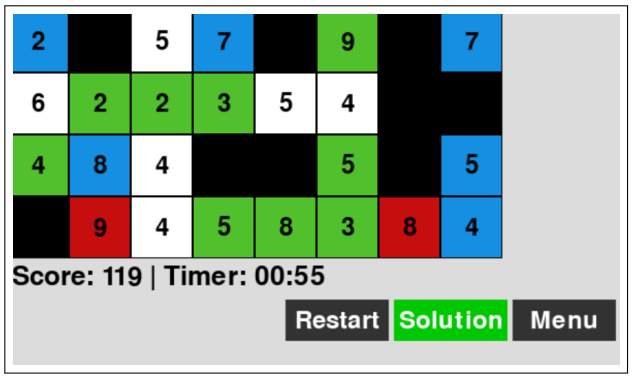

Figure 6: Game screen

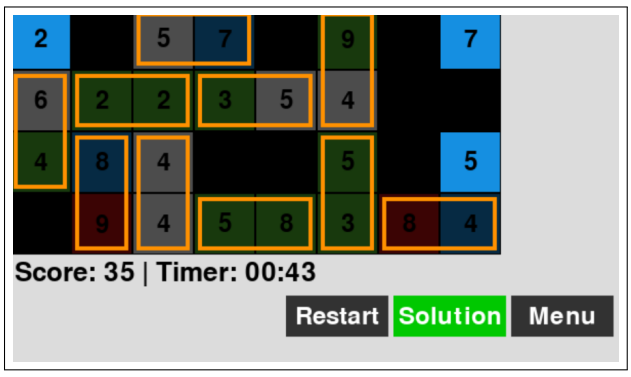

Figure 7: A solution (classic rules)

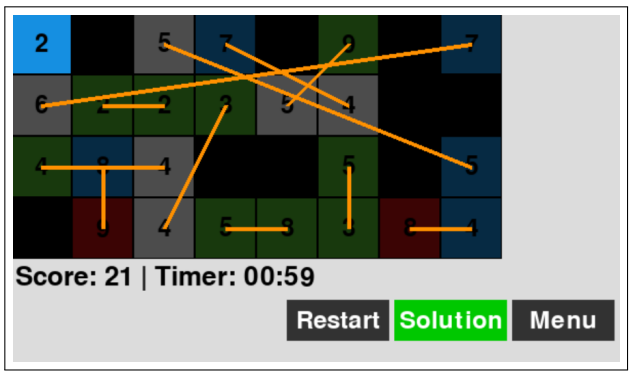

Figure 8: A solution (”no adjacency” rule)

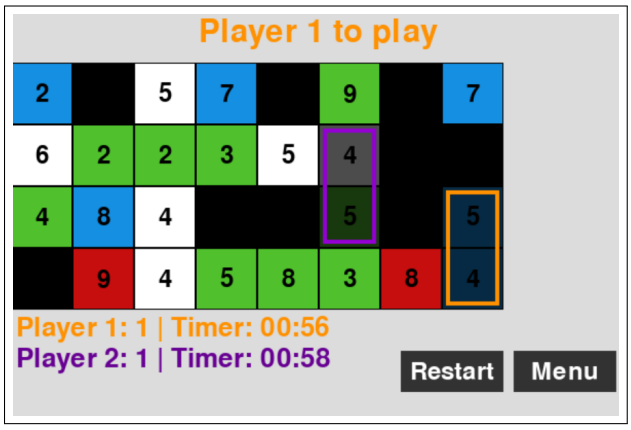

Figure 9: Two players mode

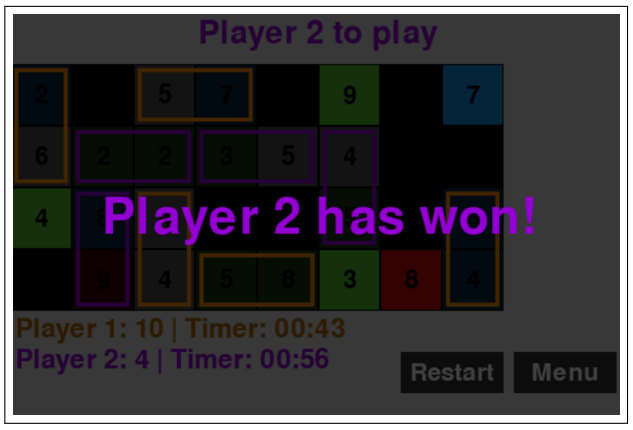

Figure 10: Game end screen

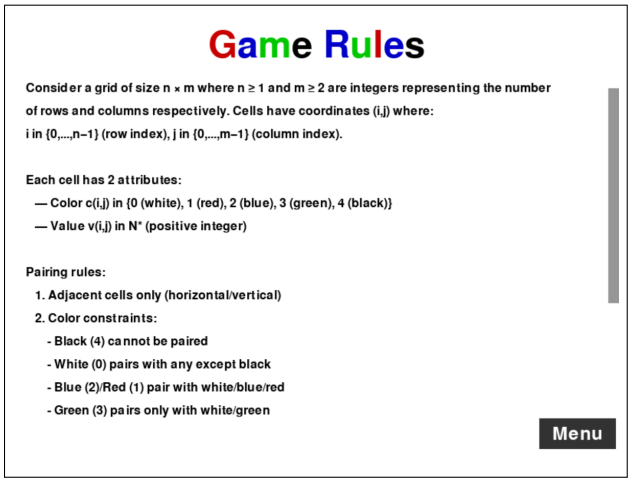

Figure 11: Game rules

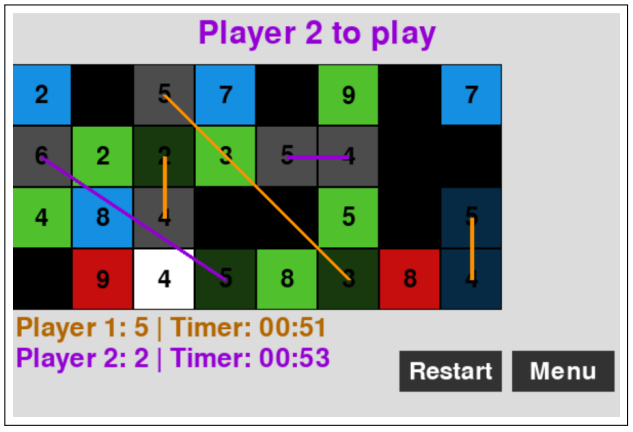

Figure 12: Two player mode ("no adjacency" rule)