Solving Flow Free Puzzles with Non-Planar Path Intersections via Backtracking: An Algorithmic Extension to the Bridges Variant

A Graph-Decomposition and Backtracking Approach for the Analysis of Non-Planar Numberlink Routing Problems

Muhammad Iqbal Raihan – 13524011

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: 13524011@std.stei.itb.ac.id, miqbalr001@gmail.com

***Abstract*—Flow Free: Bridges is a grid-based puzzle that allows path crossings at designated bridge tiles, introducing non-planar layouts that are impossible in the base game. We formalize the Bridges rules using a graph-theoretic dual-node model that splits each bridge into horizontal and vertical virtual sub-nodes, transforming the layout into a vertex-disjoint path-routing problem. Based on this model, we design a depth-first backtracking solver in Go, optimized with constraint-based pruning strategies, including an online connectivity check via Breadth-First Search. Empirical evaluations on handcrafted test instances demonstrate that the solver resolves small puzzles rapidly, but exhibits exponential search space growth in the worst case, aligning with the NP-complete complexity of numberlink routing.**

***Keywords*—backtracking; Flow Free; Numberlink; bridges; constraint satisfaction; non-planar routing; graph decomposition**

I. Introduction

Flow Free, originally developed by Big Duck Games [4], is a modern adaptation of the classical Japanese logic puzzle known as Numberlink. The puzzle is played on a two-dimensional grid of size $N \times M$, where a set of colors is defined, and each color is associated with exactly one pair of endpoints located on the grid. The objective of the puzzle is to connect each pair of endpoints with a continuous, simple path consisting of orthogonal grid segments. Furthermore, the puzzle requires that every non-blocked cell in the grid must be occupied by exactly one color path, and no two paths may overlap or share a cell.

From a computational standpoint, Flow Free is a highly constrained routing problem. The underlying routing problem is NP-complete in general [6, 2]. Finding a solution to the puzzle is equivalent to finding a set of vertex-disjoint paths that span the entire grid. The requirement that every grid cell must be filled adds a global density constraint, converting the puzzle from a simple path-finding problem into a specialized Hamiltonian path cover problem, which is known to be computationally hard.

The introduction of the Bridges variant of Flow Free [4] changes the topological landscape of the puzzle. A bridge tile is a special cell that allows two paths of different colors to cross over each other. Specifically, one path is allowed to transit horizontally across the tile, while another path transits vertically. Because paths can cross, the routing problem is no longer restricted to planar graphs. This non-planarity introduces both opportunities and challenges: while it allows solutions that are topologically impossible in standard Flow Free, it also expands the search space by adding multiple possible states to the bridge coordinates.

Flow Free serves as a representative benchmark for constraint satisfaction problems (CSPs) and state-space search analysis. The goal of this paper is to design, model, and analyze a formal backtracking solver tailored for the Flow Free: Bridges variant.

Our primary contributions are fourfold:

1. We define a formal graph model, the dual-node bridge model, which represents bridge tiles as two distinct, non-interacting horizontal and vertical sub-nodes.
2. We design a depth-first backtracking search algorithm augmented with several pruning heuristics, including an online connectivity verification step using Breadth-First Search (BFS) to identify dead states early.
3. We develop a fully functioning solver implementation in the Go language, offering both a high-performance command-line interface (CLI) and a web-based user interface (UI) for interactive solver visualization.
4. We evaluate the solver's empirical performance, measuring nodes expanded, backtrack counts, and search depth on planar and non-planar test cases to quantify the effectiveness of our bounding functions.

The remainder of this paper is organized as follows: Section II reviews standard Flow Free and outlines the Bridges variant rules alongside related work; Section III defines the math-

ematical formulation of the slot graph and the bounding functions; Section IV describes the backtracking algorithm and pseudocode; Section V discusses the implementation details in Go; Section VI conducts a formal complexity analysis; Section VII presents the experimental evaluation and results; and Section VIII concludes the paper and discusses directions for future work.

II. Problem Definition and Related Work

In this section, we present the formal definitions of both standard Flow Free and the Bridges variant. We also outline how bridge cells are represented in our graph model and review existing academic and practical literature on Numberlink solvers.

A. Standard Flow Free Formulation

Let the puzzle board be represented by a grid of dimensions $N \times M$. Let W be the set of coordinates containing obstacles or walls where no path may enter. Let C be the set of distinct colors present in the puzzle. For each color $c \in C$, there is a pair of distinct endpoints denoted by $e_c^1, e_c^2 \in \{0, \ldots, N-1\} \times \{0, \ldots, M-1\} \setminus W$. A path for color c is defined as a sequence of grid coordinates $P_c = (v_0, v_1, \ldots, v_l)$ such that:

1. $v_0 = e_c^1$ and $v_l = e_c^2$;
2. for all $i \in \{0, \ldots, l-1\}$, the coordinates v_i and v_{i+1} are orthogonally adjacent, meaning if $v_i = (r_i, c_i)$ and $v_{i+1} = (r_{i+1}, c_{i+1})$, then $|r_i - r_{i+1}| + |c_i - c_{i+1}| = 1$;
3. the path is simple, meaning $v_i \neq v_j$ for all $i \neq j$ (no self-intersection);
4. no coordinate in the path belongs to the wall set W.

The global constraints for standard Flow Free are:

1. Path Disjointness: For any two distinct colors $c_1, c_2 \in C$, their respective paths P_{c_1} and P_{c_2} must be disjoint, i.e., $P_{c_1} \cap P_{c_2} = \emptyset$.
2. Grid Coverage: The union of all color paths must cover every playable cell in the grid:

$$\bigcup_{c \in C} P_c = (\{0, \ldots, N-1\} \times \{0, \ldots, M-1\}) \setminus W. \tag{1}$$

B. Bridges Variant Rules

The Bridges variant relaxes the path disjointness constraint specifically at designated bridge cells. A bridge cell coordinate is denoted as $b \in B \subseteq (\{0, \ldots, N-1\} \times \{0, \ldots, M-1\}) \setminus W$. The rules governing bridge cells are formalized as follows:

1. Endpoints cannot be located on bridge cells: $\forall c \in C$, $\{e_c^1, e_c^2\} \cap B = \emptyset$.
2. A path entering a bridge cell horizontally (i.e., along the row axis) must utilize the horizontal track and continue horizontally: if $v_i = (r, c-1)$ and $v_{i+1} = (r, c) \in B$, then the next node in the path must be $v_{i+2} = (r, c+1)$ (or vice versa). No vertical exits or turns are permitted.
3. A path entering a bridge cell vertically must utilize the vertical track and continue vertically: if $v_i = (r-1, c)$ and $v_{i+1} = (r, c) \in B$, then the next node in the path must be $v_{i+2} = (r+1, c)$ (or vice versa).
4. Up to two distinct paths of different colors $c_1 \neq c_2$ may occupy the same bridge cell coordinate $b \in B$, provided that one path uses the horizontal track and the other uses the vertical track.

C. Dual-Node Bridge Decomposition Model

To model the Bridges variant without introducing complex state variables or allowing cells to hold lists of colors, we propose a graph decomposition model. Each normal grid cell $(r, c) \notin B$ is modeled as a single vertex in a search graph. Each bridge cell $(r, c) \in B$ is split into two virtual vertices:

1. A horizontal virtual vertex $v_H(r, c)$ which connects only to the adjacent cells on the same row: $(r, c-1)$ and $(r, c+1)$.
2. A vertical virtual vertex $v_V(r, c)$ which connects only to the adjacent cells on the same column: $(r-1, c)$ and $(r+1, c)$.

By decomposing the bridges, the search space becomes a standard graph routing problem where every vertex (including virtual bridge vertices) is occupied by at most one color. This transformation simplifies the backtracking solver logic significantly, as the same pruning rules can be applied uniformly across all vertices in the graph.

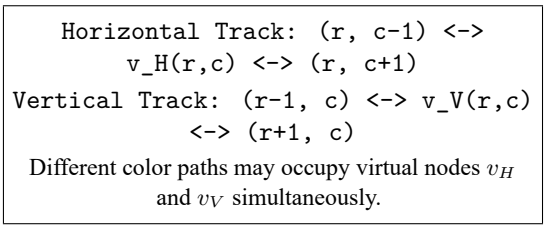

Figure 1: Bridge cell decomposition into horizontal and vertical sub-nodes, preventing intersection except through defined tracks.

D. Related Work

Numberlink puzzles have been analyzed extensively in computer science literature [1, 2]. The problem of finding disjoint paths in a grid is a classical routing problem. While standard Numberlink does not require filling every cell, Flow Free does, which mathematically links it to the Hamiltonian path problem. Solvers for Flow Free typically fall into three categories:

1. Boolean Satisfiability (SAT) and Integer Linear Programming (ILP) Encodings: These approaches translate the

grid constraints and path continuity constraints into a set of boolean clauses or linear inequalities. Solvers such as Glucose or Gurobi are then used to find a solution. Torvaney [2] and Goldman [3] demonstrate that SAT encodings can solve large Flow Free instances very quickly due to advanced clause learning.

2. Heuristic Search Algorithms: Algorithms such as A* or Greedy Best-First Search use heuristics (e.g., cell degrees, bottleneck detection) to guide path expansion. These are effective for large grids but require complex heuristic tuning.

3. Backtracking and Constraint Satisfaction: This is a fundamental approach that recursively builds paths, utilizing forward checking or arc consistency to prune dead ends. While slower than SAT on massive boards, backtracking is highly transparent, has a minimal memory footprint, and provides clear performance metrics suitable for analyzing algorithm complexity.

III. Mathematical Formulation

To build a rigorous foundation for our backtracking solver, we define the search space and pruning criteria as a formal Constraint Satisfaction Problem (CSP).

A. The Slot Graph

Let $V_{\text{grid}} = \{0, \ldots, N-1\} \times \{0, \ldots, M-1\} \setminus W$ be the set of playable cells on the board. The slot graph is defined as an undirected graph $G = (S, E)$, where S is the set of slots (search variables) and E is the set of edges (valid transitions). The slot set S is constructed as:

$$S = \{(r, c, \tau) \mid (r, c) \in V_{\text{grid}},\ \tau \in T(r, c)\}, \tag{2}$$

where the track set $T(r, c)$ is defined as:

$$T(r, c) = \begin{cases} \{H, V\} & \text{if } (r, c) \in B, \\ \{N\} & \text{otherwise.} \end{cases} \tag{3}$$

The total number of slots is $|S| = |V_{\text{grid}}| + |B| = NM - |W| + |B|$.

The edge set $E \subset S \times S$ represents valid movements between slots. For any two slots $u = (r_1, c_1, \tau_1)$ and $v = (r_2, c_2, \tau_2)$, an edge $(u, v) \in E$ exists if and only if they are adjacent under the following movement rules:

- If $\tau_1 = N$ and $\tau_2 = N$, then $(u, v) \in E$ iff $|r_1 - r_2| + |c_1 - c_2| = 1$.
- If $\tau_1 = N$ and $(r_2, c_2) \in B$, then:
  - $(u, v) \in E$ with $\tau_2 = H$ iff $r_1 = r_2$ and $|c_1 - c_2| = 1$ (horizontal entry).
  - $(u, v) \in E$ with $\tau_2 = V$ iff $c_1 = c_2$ and $|r_1 - r_2| = 1$ (vertical entry).
- If $(r_1, c_1) \in B$ and $(r_2, c_2) \in B$, then:
  - $(u, v) \in E$ with $\tau_1 = \tau_2 = H$ iff $r_1 = r_2$ and $|c_1 - c_2| = 1$.
  - $(u, v) \in E$ with $\tau_1 = \tau_2 = V$ iff $c_1 = c_2$ and $|r_1 - r_2| = 1$.

This formulation encapsulates the bridge routing rules directly into the graph topology. Paths are prevented from turning on bridge cells because there are no edges connecting (r, c, H) to (r, c, V), nor are there edges connecting horizontal neighbors to the vertical slot of a bridge.

B. Assignments and Constraints

Let $C = \{1, 2, \ldots, k\}$ be the set of colors. Each color $i \in C$ has two designated endpoints $e_i^1, e_i^2 \in S$. Because endpoints cannot be on bridges, their tracks are always N. A partial assignment is a function $\alpha : S \to \{0, 1, \ldots, k\}$, where:

$$\alpha(s) = \begin{cases} i & \text{if slot } s \text{ is occupied by color } i \in C, \\ 0 & \text{if slot } s \text{ is empty.} \end{cases} \tag{4}$$

For a partial assignment α, we define the active path segment for color i as a subgraph $G_i(\alpha) = (V_i(\alpha), E_i(\alpha))$ where:

$$V_i(\alpha) = \{s \in S \mid \alpha(s) = i\}, \tag{5}$$

and $E_i(\alpha) = \{(u, v) \in E \mid u, v \in V_i(\alpha)\}$. A complete assignment α is a valid solution if it satisfies the following conditions:

1. Full Coverage: $\forall s \in S, \alpha(s) \neq 0$.
2. Path Continuity: For each color $i \in C$, the subgraph $G_i(\alpha)$ is a simple path graph connecting e_i^1 and e_i^2. This means:
   - The degree of e_i^1 and e_i^2 in $G_i(\alpha)$ is exactly 1.
   - The degree of any other slot $s \in V_i(\alpha) \setminus \{e_i^1, e_i^2\}$ is exactly 2.
   - The subgraph $G_i(\alpha)$ is fully connected.

C. Bounding and Pruning Functions

In depth-first backtracking, we check a set of bounding functions to determine whether a partial assignment α can possibly be completed. If any bounding function evaluates to false, we prune the current branch. We define three primary bounding functions:

1. **Occupancy Constraint (B_{occ})**: Ensures that no slot is assigned to multiple colors. Since $\alpha(s)$ is a single-valued function, this is structurally guaranteed.
2. **Endpoint Integrity (B_{ep})**: A path for color i cannot enter a slot s which is the endpoint of a different color $j \neq i$.
3. **Graph Connectivity (B_{conn})**: Let $S_0(\alpha) = \{s \in S \mid \alpha(s) = 0\}$ be the set of unassigned slots. Let $G_0(\alpha)$ be the subgraph of G induced by $S_0(\alpha)$. For each color $i \in C$ that has not yet completed its path (i.e., its active endpoint is not adjacent to e_i^2), let u_i be the current head of the path for color i, and let e_i^2 be the target endpoint. If u_i and e_i^2 are not in the same connected component of $G_0(\alpha) \cup \{u_i, e_i^2\}$, then the puzzle is unsolvable from this state. We verify this condition by running a Breadth-First Search (BFS) starting from u_i within the empty slots $S_0(\alpha)$ to verify that e_i^2 is reachable.

The composite bounding function is defined as:

$$B(\alpha) = B_{\text{occ}}(\alpha) \wedge B_{\text{ep}}(\alpha) \wedge B_{\text{conn}}(\alpha). \tag{6}$$

IV. Algorithm Design

Our solver implements a structured depth-first search (DFS) over the space of partial assignments, following the backtracking framework described in [5]. The algorithm operates by routing one color at a time. This choice reduces the branching factor compared to cell-by-cell assignment, as paths grow continuously.

A. Color-Ordered State Space Exploration

The search order is determined by a permutation of colors $\pi = (\pi_1, \pi_2, \ldots, \pi_k)$. The solver processes colors in this order. To route color π_t:

1. The algorithm starts at the first endpoint $e^1_{\pi_t}$.
2. It recursively extends the path to adjacent unassigned slots.
3. When the path reaches the target endpoint $e^2_{\pi_t}$, the subproblem for color π_t is complete. The solver then recurses to route color π_{t+1}.
4. If no valid path can be formed, the algorithm backtracks to the previous color's path and attempts to route it differently.

B. Heuristics and Search Guidance

To accelerate search, we employ two key heuristics:

1. **Color Ordering**: The order π can significantly influence execution time. By default, the solver tests permutations. Colors with longer minimum path distances or restricted endpoint neighborhoods are routed first, as they impose tighter constraints on the remaining space.
2. **Look-Ahead Neighbor Sorting**: When expanding the path for color c from its current slot u, the adjacent slots $v \in N(u)$ are sorted based on their Manhattan distance to the target endpoint e^2_c. Specifically:

$$D(v, e^2_c) = |r_v - r_{\text{target}}| + |c_v - c_{\text{target}}|. \tag{7}$$

Slots closer to the target are explored first. This acts as a search heuristic to find a connecting path quickly, reducing the average depth at which successful connections are made.

C. Last-Color Hamiltonian Constraint

The final color π_k in the sequence is subject to a strict bounding rule. Since every playable slot must be filled in a valid solution, the final color's path must cover all remaining empty slots in the grid. Thus, when routing color π_k:

- The solver is only allowed to enter the target endpoint $e^2_{\pi_k}$ if there are no other empty slots remaining in the grid (or exactly one, if the target endpoint itself is counted).
- Entering $e^2_{\pi_k}$ early while $|S_0(\alpha)| > 0$ represents a dead node, as those remaining empty slots can never be filled by any other path.

This rule effectively transforms the routing of the last color into a Hamiltonian Path problem on the subgraph of remaining empty slots.

D. Recursive Solver Implementation

The core logic of the solver is formulated in the recursive function presented in Listing 1. The state is maintained via a simple array of assignments, which allows $O(1)$ updates and rollbacks.

Listing 1: Detailed backtracking and path extension loop in Go.

```
// walk attempts to extend the path for the current
     color.
func walk(color int, start, end Slot, cur Slot,
    prev Slot, colorIdx int, depth int) bool {
    // Record current step in the assignment array
    assign(cur, color)
    expandedNodes++

    // Check if we reached the target endpoint
    if cur == end {
        if colorIdx == totalColors - 1 {
            // Last color must fill all remaining
                 slots
            if allFilled() {
                return true
            }
            unassign(cur)
            backtracks++
            return false
        }
        // Recurse to solve the next color in the
             sequence
        if solveColor(colorIdx + 1) {
            return true
        }
        unassign(cur)
        backtracks++
        return false
    }

    // Sort neighbors based on Manhattan distance
         heuristic
    nextSlots := getSortedNeighbors(cur, end)

    for _, n := range nextSlots {
        // Enforce the Hamiltonian constraint for
             the last color
        if colorIdx == totalColors - 1 && n == end
             && emptySlotsCount() > 1 {
            continue
        }

        // Check structural occupancy and endpoint
             constraints
        if !canMove(color, n) {
            continue
        }

        // Execute recursive step
        if walk(color, start, end, n, cur, colorIdx
             , depth+1) {
            return true
        }
    }
```

```
    // Backtrack and restore state
    unassign(cur)
    backtracks++
    return false
}
```

Each recursive call increments the *nodes expanded* counter; each failed return increments *backtracks*. Maximum recursion depth records the longest partial path attempted.

E. Pruning Rules Summary

Table 1 summarizes the bounding checks performed during path traversal.

Table 1: Pruning rules applied during search tree traversal

Pruning Rule	Logical Condition / Action
Occupancy Rule	Prunes paths entering already assigned slots
Foreign Endpoint Rule	Prevents stepping on active endpoints of other colors
Bridge Axis Rule	Restricts bridge entry to orthogonal exit tracks only
Connectivity Rule	BFS ensures remaining targets remain reachable
Last-Color Rule	Blocks early termination of the final path

V. Implementation

The complete software package is developed in Go 1.22, structured to support clean separation of concerns and ease of benchmarking. The repository is organized under the directory ‘flow-free-bridge-solver/‘.

A. Codebase Architecture

The codebase is divided into two primary internal packages and two entry point applications:

- `internal/puzzle`: Contains data structures representing coordinates, slots, and the puzzle board. It includes the logic for parsing raw input files, validating structural integrity, and exporting the grid as ASCII tables or JSON payloads.
- `internal/solver`: Implements the core backtracking logic, neighbor generator, sorting heuristics, bounding functions (including the BFS connectivity check), and execution statistics collectors.
- `cmd/flowcli`: A command-line application that allows users to solve individual puzzles, run automated test suites, verify solutions, and output benchmarks in standard formats (e.g., CSV).
- `cmd/flowweb`: A web server that serves a responsive, modern HTML/CSS frontend and exposes a REST API (`POST /api/solve`) that processes puzzle descriptions and returns step-by-step statistics and solutions.

B. Puzzle File Specification

Puzzles are represented using a simple plain-text format. This allows easy creation and modification of test files. Listing 2 shows a sample puzzle specification.

Listing 2: Puzzle specification format for a 5x5 grid with a bridge.

```
# Dimensions: rows cols
5 5
# Colors: Name r1 c1 r2 c2
R 0 2 4 2
G 2 0 2 4
# Bridge coordinates: X row col
X 2 2
# Wall coordinates: W row col (optional)
# W 1 1
```

C. Interactive Interfaces

Figures 2 and 3 show the user interfaces of the web utility. Users can paste a puzzle layout into the text area, load it to render the initial state, and solve it instantly to view the visual paths and search statistics. The command-line output is displayed in Figure 4.

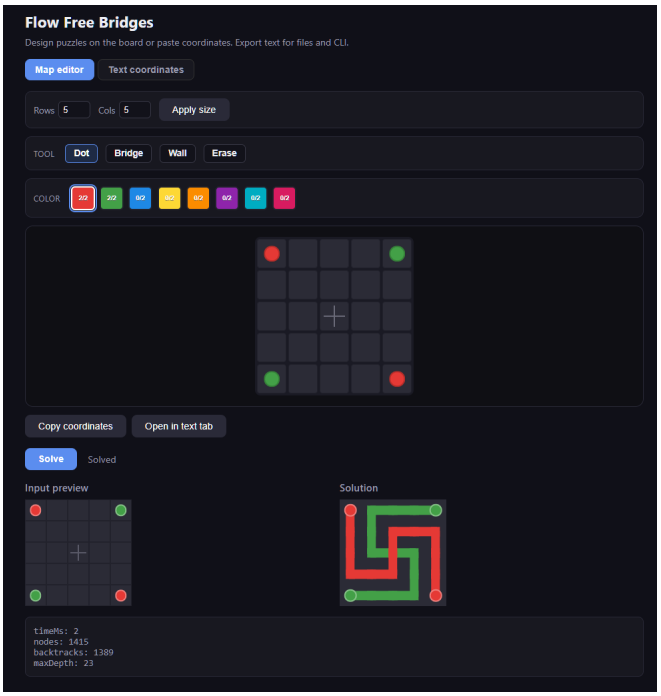

Figure 2: Map editor tab of the web UI with the default `bridge_5x5` puzzle loaded before solving.

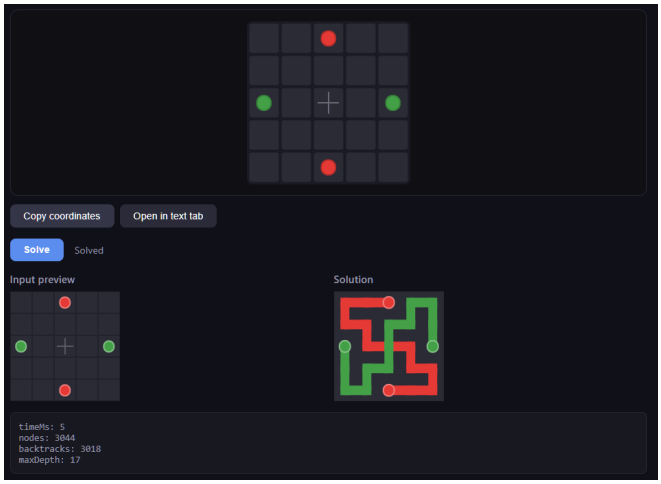

Figure 3: Web UI after solving, showing the input preview, pipe-rendered solution grid, and search statistics.

Figure 4: Terminal output of `flowcli solve puzzles/bridge_5x5.txt`, including the solved ASCII grid and search metrics.

VI. Complexity Analysis

We analyze the time and space complexity of our backtracking solver, evaluating the theoretical limits and the effects of our pruning strategies.

A. Time Complexity

Let $n = |S|$ be the total number of slots, and let $k = |C|$ be the number of colors. In the worst-case brute-force scenario, each slot can be assigned to any of the k colors or left empty (value 0). The size of this search space is:

$$\mathcal{O}\left((k+1)^n\right). \tag{8}$$

Since our algorithm builds paths sequentially, we can model the search as growing k paths. At each step, a path can extend in at most 3 orthogonal directions (excluding the cell it came from). Therefore, the branching factor of the search tree is bounded by 3. The worst-case search tree size is:

$$\mathcal{O}\left(3^n\right). \tag{9}$$

Pruning heuristics significantly reduce the average-case complexity, but they do not alter the worst-case asymptotic bounds. For example, the BFS connectivity check B_{conn} requires traversing the remaining empty slots $S_0(\alpha)$, which takes $\mathcal{O}(n)$ time. Thus, the time spent at each search node is $\mathcal{O}(n)$. Because the solver tries permutations of color orderings to find a feasible sequence, we multiply the search time by the number of permutations explored. In the worst case, this adds a factor of $k!$, leading to a total time complexity of:

$$\mathcal{O}\left(k! \cdot n \cdot 3^n\right). \tag{10}$$

While $k!$ grows rapidly, in practical Flow Free puzzles, k is small ($k \leq 6$), making the color permutation search manageable.

B. Space Complexity

The space complexity is determined by the size of the recursion stack and the memory required to maintain the assignment state.

1. **Assignment State**: The solver maintains the assignment array α of size n, requiring $\mathcal{O}(n)$ space.
2. **Recursion Stack**: The maximum recursion depth is bounded by the number of slots n. Each stack frame stores local variables (such as the current slot, neighbor lists, and loop counters), taking $\mathcal{O}(1)$ space per frame. Thus, the stack memory is bounded by $\mathcal{O}(n)$.
3. **BFS Queue**: The BFS connectivity check uses a queue that holds at most n slots, requiring $\mathcal{O}(n)$ helper space.

Therefore, the total space complexity is strictly linear:

$$\mathcal{O}(n) = \mathcal{O}(NM + |B|). \tag{11}$$

Compared with SAT encodings [2, 3], backtracking uses $\mathcal{O}(n)$ memory instead of storing large learned clause sets, and exposes each dead node as a concrete constraint violation—useful for pedagogy even when SAT solvers are faster on large instances.

VII. Experiments and Results

We conduct empirical evaluations to analyze the performance of our solver under different configurations.

A. Experimental Setup

The experiments are conducted on an AMD Ryzen 7 processor with 16 GB of RAM, running Windows 11. The solver is compiled using Go 1.22.1 with optimizations enabled. We run the benchmarks using the command:

```
flowcli bench puzzles/
```

We record the number of nodes expanded, the number of backtracks performed, the maximum recursion depth, and the total execution time in milliseconds (ms).

B. Discussion of Benchmarks

Table 2 shows the performance metrics on four representative puzzle instances.

- `tiny_2x2`: A simple 2×2 grid with two colors. The solver finds the solution immediately in 4 node expansions with 0 backtracks, indicating that the heuristics guided the search directly to the goal.
- `classic_3x3`: A 3×3 grid with three colors. The solver resolves the puzzle in 9 node expansions and 0 backtracks, representing a linear-time solution.

- `bridge_5x5`: A 5×5 grid containing one bridge coordinate. The puzzle requires a crossing of two paths. Because of the bridge, the slot count is 26. The solver expands 3,044 nodes and performs 3,018 backtracks, taking 4 ms. The high number of backtracks is a direct consequence of the Hamiltonian path constraint on the final color, which forces exhaustive search over the remaining cells to ensure complete grid coverage.
- `unsolvable_3x3`: A 3×3 grid designed to be unsolvable. The connectivity bounding function B_{conn} detects that the endpoints of a color are disconnected in the first step, pruning the entire search tree immediately and returning failure after expanding only 1 node.

Table 2: Benchmark results (representative run)

Puzzle	Solved	Nodes	BT	Depth	ms
`tiny_2x2`	yes	4	0	2	0
`classic_3x3`	yes	9	0	3	0
`bridge_5x5`	yes	3044	3018	17	4
`unsolvable_3x3`	no	1	1	1	0

BT = backtracks.

The benchmark results confirm that the BFS connectivity check is highly effective at terminating search on unsolvable branches, preventing the solver from entering deep search trees that cannot yield a valid solution. However, on larger solvable puzzles with bridges, the search space grows exponentially, which is consistent with the NP-complete classification of the routing problem [6]. Figure 5 reports the CSV output from the batch benchmark command.

```
PS D:\Ihan\Sekolah\.uni\sem 4\Strategi Algoritma\makalah\m1\flow-free-bridge-solver> ./flowcli bench puzzles/
file,rows,cols,colors,solved,nodes,backtracks,maxDepth,ms
bridge_5x5.txt,5,5,2,yes,3044,3018,17,2
classic_3x3.txt,3,3,3,yes,9,0,3,0
tiny_2x2.txt,2,2,2,yes,4,0,2,0
unsolvable_3x3.txt,3,3,3,no,1,1,1,0
```

Figure 5: Batch benchmark output produced by `flowcli bench puzzles/`, listing nodes expanded, backtracks, and runtime per puzzle file.

VIII. Conclusion and Future Work

In this paper, we formalized the Flow Free: Bridges puzzle as a search problem over a slot graph $G = (S, E)$. By decomposing each bridge cell into separate horizontal and vertical virtual slots, we successfully modeled non-planar path crossings within a standard graph framework. This dual-node bridge model allows the use of uniform backtracking search and pruning strategies.

To solve the puzzle, we designed a depth-first backtracking solver equipped with occupancy, endpoint, and BFS-based connectivity constraints. The solver also enforces a Hamiltonian constraint on the final routed color to ensure complete grid coverage. Experimental results demonstrate that the solver is highly efficient on small and medium grids, resolving simple configurations in milliseconds and pruning unsolvable boards instantly.

Future work will focus on:

1. **Constraint Propagation**: Implementing algorithms like Forward Checking or Arc Consistency (AC-3/AC-4) to pre-emptively reduce the domains of unassigned slots, which would further prune the search tree.
2. **Variable Ordering Heuristics**: Utilizing the Minimum Remaining Values (MRV) heuristic to dynamically select which color or cell to route next.
3. **Parallelization**: Distributing the search over multiple CPU cores by parallelizing the exploration of different color permutations.

Attachment

[1] GitHub Repository: https://github.com/Gixgine-budi/flow-free-bridge-solver

Acknowledgment

The author thanks the academic mentors at the School of Electrical Engineering and Informatics, Institut Teknologi Bandung, for their guidance on the algorithm design and analysis frameworks utilized in this study.

References

[1] D. Dam, "Solving Flow Free, in math," *Dante Dam Blog*, 2020. [Online]. Available: https://dantedam.com/blog/flow-free.

[2] Torvaney, "Flow Free: A SATisfying solution," *Torvaney's Personal Projects*, 2018. [Online]. Available: https://torvaney.github.io/projects/flow-solver.html.

[3] S. Goldman, "Solving Flow Free with Python and Rust," *Samuel Goldman Technical Blog*, 2021. [Online]. Available: https://samueltgoldman.com.

[4] Big Duck Games, "Flow Free: Bridges," mobile puzzle game, 2013.

[5] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*. New York, NY, USA: Computer Science Press, 1998.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, USA: W. H. Freeman & Co., 1979.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Muhammad Iqbal Raihan
13524011