An Algorithmic Approach to Tile Placement in the Pipe Work Board Game

Jennifer Khang - 13524110
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: jenniferkhang07@gmail.com, 13524110@std.stei.itb.ac.id

***Abstract*—Pipe Work is a board game best known for its tile-placement and construction mechanics, in which players are required to place pipe tiles on a board to form an interconnected pipeline systems while maximizing resources they have. The objective of the game is to connect pipe segments to their designated terminal points while satisfying the constraints imposed by the puzzle. This study models the Pipe Work puzzle as a state-space search problem by reducing it to a Flow Free game formulation, where each cell is assigned a color identifier rather than a tile type, allowing all facility to be sequentially paired. While the simultaneous multi-pair routing problem is NP-complete in the general case, the fixed 5×5 board keeps the search tractable for exact methods. Three algorithms are implemented and compared which are Brute Force, Plain Backtracking (BT), and Backtracking with Constraint Pruning (BT+CP). Results show that BT+CP expands significantly fewer nodes than BT and Brute Force while maintaining solution validity, demonstrating that constraint-aware pruning reduces the search space for the puzzle without sacrificing correctness.**

Keywords—Backtracking, Constraint Pruning, State-Space Search, Tile Placement, Graph Connectivity, Pipe Work, NP-Complete

I. INTRODUCTION

Puzzle-solving problems have long served as a benchmark for evaluating search algorithms. Cases such as 8-Puzzle, Sudoku, and Rubik's Cube have been extensively studied due to their well-defined rules, challenging solution spaces, and suitability for analysis in an algorithmic way. Various approaches have been proposed to address these problems, including exhaustive search methods, greedy strategies, backtracking, branch-and-bound algorithms, and path-finding optimization methods. Despite the overbearing amount of research, each puzzle has their own unique way to solve, specified by their own environments. These differences in environment have created numerous openings for game enthusiasts to solve games that have yet to come in a creative and adaptive way.

Fig. 1. Gameplay of Pipe Work. *(Source: https://youtu.be/Qn0q8bC10cs?si=9rIy6mPfq4ESTlvL)*

The reason for this study is quite simple. A game session at a board café with friends had inspired author to find an algorithmic way of finding a solid solution, specifically for a notorious game called Pipe Work. Pipe Work itself is a board game published by Broadway Toys LTD, Gemblo Company. It's a creative game that can be played with the amount of 1 to 4 people, where gameplay includes placing pipes or tubes in each grid. Similar to the nature of puzzle-solving games in general, the game requires players to identify a valid solution among numerous possible configurations. As the number of possible tile placements increases, determining an appropriate solution becomes increasingly complex, making the game suitable for analysis using search and optimization techniques. Until now, there is no known consistent algorithm to solve those board configurations, which can be called "maps". Although, it might be similar to cases such as collision avoidance in a networking problem, which is in the interest of the author.

Overall, this paper models the Pipe Work puzzle as a state-space search problem, where each state represents a partial board configuration and each action corresponds to the placement of a pipe tile. Two backtracking-based strategies — Backtracking (BT) and Backtracking with Constraint Pruning (BT+CP) — are evaluated on a collection of puzzle instances to compare their effectiveness in constructing valid solutions for the configurations. Performance is assessed by using nodes expanded, number of node expansion and execution time. Through this comparative study, the paper aims to evaluate the effectiveness of constraint pruning or better known as bounding function in reducing the search space and improving the efficiency of solving Pipe Work's puzzles.

II. Related Work

Search algorithms for tile-placement puzzles have been studied thoroughly. Despite differences in rules, Pipe Work shares several gameplay mechanics Pipe Dream or Pipe Mania, a game with similar concepts but different premises. Both game are originally developed by The Assembly Line and released in 1989, where players race to lay down a sequence of straight, corner, and crossover pipe segments to channel a flow between two points before time runs out. Pipe Work modernizes this concept into a simultaneous multiplayer board game in which each player races on their own 5×5 grid to connect pairs of colored tokens with pipe tiles before the others do. The following is some research or source that might be related to this study.

1) "Resolvendo Pipe Mania como Planejamento" (Banci, Guilherme A. D., 2022)[3]. A thesis that mainly explain about PipeMania solution by finding the optimal route to connect source to destination, end-to-end. This research help as a baseline on what author should do in the next few days, it might not be related but it help fundamental understanding.

2) Zucker (2016) implemented a fast Flow Free solver in C using best-first search augmented with domain-specific validity checks including dead-end detection, stranded color analysis, and chokepoint identification, demonstrating that constraint-aware pruning reduces node expansions by over an order of magnitude compared to uninformed search [5].

There are some concerning findings that the author found during the research. It has been stated from "It is tough to be a plumber" that Pipe Mania—which have been mentioned before this paragraph—is an NP-Complete problem [4]. It might be the same case for this board game, in a way that the solution may arrive at a non-polynomial time, as the type of tiles increases, which is ineffective. However, this situation can be diverted as the grid is still considered small in the eyes of computation machines and some tiles may be removed. From here on, I will base my hypothesis and assumption from the previous work mentioned. Alas, this is a niche topic that barely got any attention.

III. Theoretical Framework

Before beginning the research itself, there needs to be an understanding of how the game works and a few terms that will be used in this research.

A. Pipe Work Board Game

Overall, Pipe Work is played on an individual 5×5 grid per player. Over eight rounds, a card reveals where pairs of colored tokens (they are called factory) must be placed on the grid. Then, all players race to finish the board by placing pipe tiles from their hands onto the empty cells of their own board, in a way that every factory must connect with their respective counterpart color through the pipes. Each grid must be filled with either a factory or pipe, thus making the game differ from previous version like Pipe Dream that doesn't enforce this rule. Players who finish take a numbered "spanner" card in the order they finish, with the fastest correct solution rewarded the most, while an incorrectly connected board is penalized. After eight rounds, the player with the most accumulated rewards wins.

The tiles itself consist of 3 pipe tiles, which are straight segments, corner (elbow) segments, and crossover pieces. For tractability, and because any path on a 4-connected grid can be fully described using only straight and corner segments, this study restricts its tile vocabulary from crossovers, which only matter when a single cell must carry more than one pipeline through it, are left for future work.

Each configuration problem also has their own level of complexities consisting of 4 types. It is chosen depending on the current player's demands. The following are the complete complexities of each category.

a. The first category requires players to connect all four colored facilities without the use of crossover pipes, making it the most straightforward challenge.
b. The second category increases the difficulty by requiring only three facilities to be connected while one facility is excluded from the solution, effectively reducing the available routing space.
c. The third category requires all four facilities to be connected and mandates the use of crossover pipes, introducing additional routing complexity.
d. Finally, the fourth category combines both selective connectivity and mandatory crossover usage, requiring players to connect only three facilities while utilizing crossover pipes to overcome routing constraints.

This paper will only implement level 1 and 2 due to complexity using crossover pipes.

B. Graph Representation

1) Grid-Filling Actions

Let the Pipe Work board be simulated by an m × n grid. We define a graph G = (V, E) where each vertex v ∈ V corresponds to cell (i, j) of the grid, and each edge $(v_{ij}, v_{i'j'}) \in E$ represents a valid pipe connection between adjacent cells. Formally, it can be defined as follows.

$$V = \{v_{ij} \mid 0 \le i \le m, 0 \le j \le n\}$$
$$E = \{(v_{ij}, v_{i'j'}) \mid |i - i'| + |j - j'| = 1\}$$

Each cell may hold a pipe tile drawn from a finite tile set T = {t_1, t_2, ..., t_n}. Each tile t ∈ T is characterized by a connection mask μ(t) ⊂ {N, S, E, W}, indicating which of its four sides carry a pipe opening. A tile placement is valid at cell (i, j) if for every adjacent cell (i', j') connected by μ(t), a matching opening exists in the tile placed at (i', j'). This is if we focus on pipe abstraction, it may cause a large computational cost. Therefore, this problem may be reduced to another abstraction that may help increase computation time, see Section III.C for explanation.

To find the solution, we must fill the board by selecting, for each empty cell along a pipeline's route, a tile shape —

either straight or corner — and an orientation, and committing it to that cell. Once placed, a tile is fixed for the remainder of the search unless the solver backtracks past that decision. A placement is only valid if it satisfies the following constraints simultaneously.

- Continuity: for a given token pair, the sequence of placed tiles from one token to the other must form a single unbroken chain, where each tile's opening aligns exactly with the opening of its neighbor — no gaps and no mismatched ends.
- Non-overlap: no cell may be claimed by more than one pipeline, since each cell holds at most one physical tile. A pipeline belonging to one token pair can never pass through a cell already used by another pair's completed or in-progress pipeline. This is also the reason why we won't be using crossover pipes since it will ruin the algorithm the author's aiming for.

2) Deduction Strategy

Before resorting to exhaustive search, a solver can exploit the following constraints to narrow the search space by pruning.

- Dead-end detection. An empty cell is classified as a dead end if at least three of its four neighbors are either walls or occupied non-frontier cells, leaving it with at most one entry point. Such a cell can never be properly filled without stranding the pipeline, so any state containing a dead-end cell is immediately rejected.
- Reachability verification. For every unfinished color, a BFS is performed from the color's current head position to confirm its goal is still reachable through empty cells or the goal cell itself. If any unfinished color cannot reach its goal, the state is rejected immediately without further expansion.
- Free region analysis. Every contiguous region of empty cells is identified through flood-fill. A region is considered stranded if it contains no cell adjacent to any unfinished color's current head or goal position — meaning no active pipeline can enter and fill it. Any state containing a stranded region is rejected.

3) Goal State

A valid goal state is reached when every token pair has been connected by a continuous, correctly aligned chain of pipe tiles, where no two pipe tiles occupy the same cell, and no placed tile is left with an unmatched opening or open end. Once all the conditions are met simultaneously across every pair on the board, the board represents a complete solution for the configurations assigned.

C. Computational Complexity and NP-Completeness

Based on their running time, algorithms are commonly classified into two large groups. Polynomial-time algorithms have costs bounded by a polynomial function of the input size, while non-polynomial-time algorithms grow faster than any polynomial, typically exponentially. A problem is tractable if it admits a polynomial-time algorithm and intractable if it does not [1].

Complexity theory frames this around decision problems — problems whose answer is simply "yes" or "no." Class P consists of every decision problem solvable by a deterministic algorithm in polynomial time. Class NP consists of every decision problem for which a proposed solution can be verified in polynomial time once given which makes P a subset of NP. Class NP-Complete (NPC) is the set of the hardest problems within NP. Problem X is NP-Complete if (1) X is in NP, and (2) every other problem in NP can be reduced to X in polynomial time.

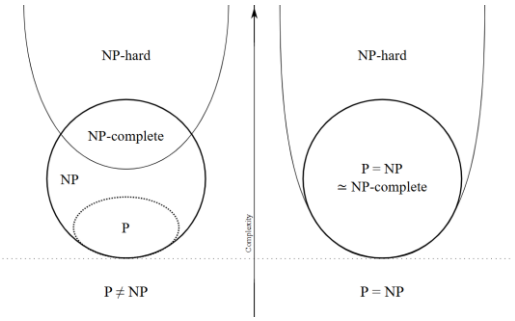

Fig. 2. Euler diagram for P, NP, NP-complete, and NP-hard set of problems. *(Source: https://en.wikipedia.org/wiki/NP_%28complexity%29)*

Pipe Work solution is simple. Given a board with k token pairs, does there exist an assignment of pipe tiles to cells such that all k pairs are simultaneously connected and no two tiles share a cell? The following are reasons why Pipe Work or similar games are considered NP-Complete and why it will affect the proposed method of this paper.

- NP membership: checking a candidate solution only requires tracing each pipeline once to confirm it connects the correct pair and that no cell is shared — both doable in a single linear pass over the board, which is polynomial in the board's size. Checking whether the grid is fully occupied is easy to be checked in a polynomial time.
- NP-hardness: stripped of tile-shape detail, this is the question of whether k terminal pairs on a grid can be connected by k vertex-disjoint paths — proven NP-complete when the number of pairs is allowed to grow with the board size [4]. Requiring physical tile realizability only adds constraints, so Pipe Work's joint routing problem inherits this hardness in the general case.

This suggests its similarity to class NP. There is a high chance that Pipe Work may be reduced to another form of problem belonging to the same class. A very good candidate that we may consider is Flow Free. The game has similar features to Pipe Work, except for tiles such as crossover pipes, such that its representations will be changed. This method will be considered if backtrack by pipe abstractions are proven too costly.

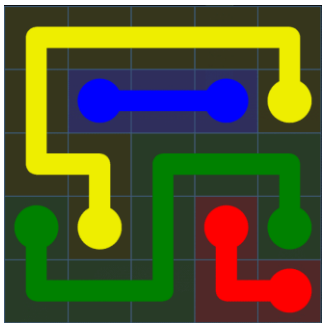

Fig. 3. Flow Free Game's 5×5 Configuration Example. *(Source: https://givemetheanswer.com/flow-free/bonus-pack/5x5-solutions/)*

For the fixed 5×5 board of the actual game, the number of pairs is small and bounded, keeping each individual routing instance tractable. However, the joint simultaneous routing problem still grows expensive in practice as pairs compete for limited cells, motivating the use of backtracking with constraint pruning rather than exhaustive enumeration, this is viable.

D. Backtracking Algorithm

Backtracking is a refinement of exhaustive search. Exhaustive search evaluates every candidate solution one by one whereas backtracking only continues exploring choices that can still lead to a valid solution, abandoning or so-called *pruning* those that cannot as early as possible. In the field of Artificial Intelligence, backtracking is also known as a child of Constraint Satisfaction Problem. This paper implements Constraint Propagation on a lower level. There are also a few considerations that need to be addressed, such as how to choose the next variable, some method includes MRV (Minimum Remaining Values), Degree, Least Constraining Value. Then, we need to check the constraint consistency. The following is some concepts explained and related to Pipe Work.

1) *Constraint Propagation*
   Constraint propagation is an inference technique used in Constraint Satisfaction Problems (CSPs) to reduce the domains of variables before or during search. By removing values that violate constraints, the search space is reduced without sacrificing completeness. This process may also produce deterministic assignments when a variable's domain is reduced to a single legal value, allowing the assignment to be made immediately before further search
2) *Minimum Remaining Values (MRV) Heuristics*
   The Minimum Remaining Values (MRV) heuristic is a variable-ordering strategy that selects the variable with the fewest remaining legal values.
3) *Manhattan Distance Heuristics*
   For search problems defined on a two-dimensional grid, the Manhattan distance is a commonly used heuristic function. Given a current position (x, y) and a goal position (x_g, y_g), the heuristic value is defined as

$$h(p) = |x - x_g| + |y - y_g|.$$

The Manhattan distance estimates the minimum number of horizontal and vertical moves required to reach the goal when movement is restricted to four directions. It is frequently used to guide searches toward promising states.

A backtracking formulation requires three components.

1) *Solution vector*
   The solution is expressed as an n-tuple $X = (x_1, x_2, \ldots, x_n)$.
2) *Generating function*
   A function $T(x_1, \ldots, x_{k-1})$ that produces candidate values for the next component x_k.
3) *Bounding function*
   A predicate $B(x_1, \ldots, x_k)$ that returns true if the partial assignment still has a chance of leading to a valid solution, and false if it violates a constraint. A true result continues the search to step k+1 while false meant discarding that branch.

All possible solutions form a rooted state space tree, where each node represents a partial solution at some stage and each edge represents one variable assignment. The search proceeds in four repeating steps, which are (1) node generation, extending the current E-node along a DFS-style path, (2) pruning, where the bounding function kills the current node if it cannot lead anywhere useful, implicitly discarding its subtree, (3) backtrack, returning to the nearest live ancestor once a dead node is reached, and (4) further exploration, generating the next untried child and repeating until a solution is found or the search space is exhausted.

IV. Proposed Method

The methodology for this research was selected to address how a specific approach can be utilized for solving the problem. We will create a simulation and compare the proposed algorithm used to one another, to prove the credibility of this paper itself. Firstly, the author had done some research and studied several algorithms that might fit the rules of this game. Before moving on to the explanation of the method, the following are some scope constraints on where and how far the author is experimenting.

a. This study will only serve the purpose to analyze how *'fast'* each algorithm may solve the problem of placing the tiles, in a way that connects each respective node. Meaning, we are finding alternative ways to solve Pipe Work.

b. This study is not about finding ways to win an actual game session, as a player's objective is to gain the most rewards across all eight rounds. This will be ending in a greedy approach by counting two largest sum resources for each reward later and is not within the scope. Rather, it isolates and measures the cost of solving each configuration grid which must be repeatedly performed within a single round.

c. All facility pairs are routed jointly in a single search pass, with every path constructed concurrently. A cell claimed by one color becomes immediately unavailable

to all others, enforcing the non-overlap constraint throughout the search without requiring sequential pair ordering.

For this study, an appropriate approach is by using state-based algorithms. Because the goal is an absolute solution rather than an approximation, algorithms that sacrifice solution validity for speed are not considered. This game is bound by rules that make it impossible to just find the optimal rule due to the fact each grid must be filled. Another cause was that the manual itself has stated a fixed solution for each configuration. Hence, the only meaningful axis of optimization is how efficiently the search reaches a goal state from the initial empty board. Backtracking is the fit, it systematically explores the space of placement decisions, pruning branches the moment a constraint is violated, and is guaranteed to find a valid solution if one exists. This is a hypothesis that author holds. For extra comparisons, author will also conduct exhaustive search as a baseline.

A. Bruteforce Approach

This algorithm is a naïve and straightforward approach. It will iterate every possibility without any special mechanism. Although simplistic, it has a backhand weakness too, such as:

- Poor Scalability. As the number of cells increases, the execution time increases exponentially, causing rapid performance degradation.
- Cache Inefficiency. Random memory access patterns.
- Redundant Calculations. No spatial locality exploitation.
- No Early Termination. This algorithm will check every possible state regardless of spatial distribution.

Analyzing the algorithm to find the complexity of this approach:

- Time Complexity: O(n).
- Space Complexity: O(1) additional space.

Bruteforce work as follows.

1. Generation. Generate every board configurations to fill all 25 cells.
2. Recursive. For each complete board configuration:
   a. Check if all pairs are connected
   b. Check if no dangling endpoints exist
   c. If it is valid, the algorithm must return the solution.
3. Basis. If there is no configuration satisfies constraints, return failure as an answer.

B. Backtrack as Main Hypothesis

We may apply backtracking to Pipe Work, such as:

- Solution vector. An ordered sequence of placement decisions, where each x_i is a tuple (cell, tile shape, orientation) extending one of the in-progress pipelines.
- Generating function. The next candidate placements are the empty cells adjacent to the current pipeline's tip, combined with each tile shape and orientation that could legally continue from there.
- Bounding function (BT). Reject only when the current cell has no unvisited, unobstructed neighbors, which means a dead-end condition with no look-ahead.
- Bounding function (BT+CP). Additionally reject any extension that, according to a BFS connectivity check, either (a) leaves the current pair's goal unreachable from the current tip, or (b) leaves any future pair's two endpoints in disconnected components of the remaining free cells.
- State space tree. The root is the empty board with placed tokens. Each level corresponds to one placement decision, and a leaf is either the goal node (all pairs validly connected) or a dead end that will be cut.

Analyzing the algorithm to find the complexity of this approach:

- Time Complexity: The time complexity of plain BT in the worst case is $O(b^d)$, where b is the branching factor (the number of valid neighboring cells at each step) and d is the depth of the solution (the total number of tile placements required to fill the board) [1]. On a 5×5 board with at most 4 directional neighbors per cell, $b \leq 4$ and $d \leq 25$, so the theoretical worst case is bounded. BT+CP reduces the effective branching factor by pruning extensions that violate reachability constraints, with an exchange cost of a BFS check of $O(V) = O(25)$ per candidate extension, ensuring the possibility of branching.
- Space Complexity: Both BT and BT+CP require O(d) space for the recursion stack, where d is the search depth. On 5×5 board, this is bound by 25 recursive calls. BT+CP additionally requires O(V) auxiliary space for the BFS queue and visited array during each reachability check. Since the board size is fixed, the overall space complexity remains O(d+V), which is effectively constant in practice.

V. EXPERIMENTAL SETUP

Programming languages chosen for implementation is Go 1.26.4, known for its development phases of the system to be fast and also allowing to design the system as modular. The initial approach modeled each cell as a typed pipe tile with an explicit connection mask, requiring the solver to track tile shape and orientation at every placement decision. This formulation proved computationally expensive due to the large branching factor introduced by multiple tile types and rotations per cell.

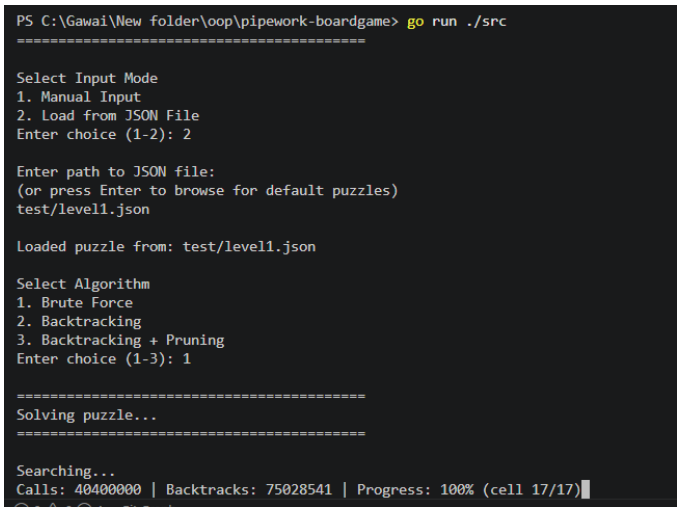

Fig. 4. First Try Failed Due to Tile Abstractions. *(Source: Authors' Personal Archive)*

The model was therefore simplified by reducing the Pipe Work board to a Flow Free representation, where each cell is assigned a color identifier rather than a tile type [5]. Under this abstraction, a valid solution requires every cell to be assigned exactly one color, every color to form a single continuous path between its terminal pair, and no two colors to share a cell. This reduction eliminates tile-shape tracking entirely while preserving all connectivity constraints relevant to the puzzle, resulting in a significantly smaller state space and more tractable search.

A. Modified Representation

The following is several data type use for this experiment.

- The board is represented as a 5×5 integer grid where each cell stores either an empty marker (0), a wall marker (-1), or a color identifier assigned to each facility pair.
- Terminal positions are pre-filled with their respective color identifiers at initialization.
- A search state consists of the current board configuration, the path taken by each color so far, a completion flag per color, and the goal position for each color. This meant that each cell is not filled with pipe tiles, but key movements whether they should go up, down, left, or right.

Data Type and State

```
BOARD:
  cells[5][5]      : integer grid
                       0   = empty
                      -1   = wall
                      >0   = color ID

  colorIDs         : map[colorName → colorID]
  colorNames       : map[colorID → colorName]
  terminals        : map[position → colorName]
  goals            : map[colorName → position]
  starts           : map[colorName → position]
  terminalCells    : map[position → bool]

SEARCH STATE:
  board            : Board
  paths            : map[colorName → []position]
  completed        : map[colorName → bool]
  goals            : map[colorName → position]

POSITION:
  row, col         : integer

PUZZLE:
  size             : 5 (fixed)
  pairs            : []FacilityPair
  walls            : map[position → bool]
```

B. Algorithm Implementation

1) Bruteforce

Bruteforce

```
function BruteForce(puzzle) → Result
  allPaths ← EnumerateAllPaths(puzzle)
  solution ← TryCombinations(allPaths, puzzle)
  if solution = NULL:
    return Result(success = false)

  return Result(success = true, solution)

end function
```

2) Backtrack

a. Implemention of Bounding Function

From Section III.B with Deduction strategy, we can infer the following constraint checking with the following.

Bounding Function

```
function PassesConstraintPruning(state, puzzle) →
       boolean

  frontier ← { currentHead(c) | c unfinished }
           ∪ { goal(c) | c unfinished }

  // 1. Dead-End Check
  for each empty cell pos in board do
    blocked ← 0
    for each neighbor n of pos do
      if n is wall: blocked++
      else if n ∉ frontier AND n is occupied:
       blocked++
    end for
    if blocked ≥ 3: return false
  end for

  // 2. Reachability Check
  for each unfinished color c do
    if NOT BFS_Reachable(currentHead(c), goal(c),
       state):
      return false
  end for

  // 3. Stranded Region Check
  visited ← {}
  for each empty cell pos in board do
    if pos ∈ visited: continue
    region, touchesFrontier ← FloodFill(pos,
       frontier)
    visited ← visited ∪ region
    if NOT touchesFrontier: return false
  end for
```

```
  return true
end function
```

b. Implementation of Backtrack

- Base Case Check. If all colors are completed and the board is fully filled, the search halts and returns the goal state.
- Constraint Propagation (BT+CP only). Before branching, the algorithm checks whether any unfinished color has exactly one valid move. If so, that move is applied automatically without creating a branch. This continues until no forced moves remain or a constraint violation is detected.
- Variable Ordering. For BT, the active color is selected in alphabetical order. For BT+CP, the Most Constrained Variable Heuristic selects the color with the fewest valid moves first, reducing the branching factor early.
- Move Generation and Pruning. Valid moves for the active color are the empty neighbors of its current head plus its goal cell if adjacent. Moves are ordered by Manhattan distance to the goal. For BT+CP, each candidate move is evaluated by the bounding function before expansion. If the bounding function returns false, the branch is pruned immediately.
- Backtrack. If all moves for the active color are exhausted without finding a solution, the function undoes the last placement and returns to the nearest ancestor with unexplored branches.

Backtrack

```
function Backtrack(state)

    if GoalReached(state) then
        return state
    end if

    color ← NextColor(state)        // alphabetical
order

    for each move in ValidMoves(state, color) do
        next ← ExtendPath(state, color, move)

        solution ← Backtrack(next)

        if solution ≠ NULL then
            return solution
        end if
    end for

    return NULL

end function
```

Backtrack with Pruning

```
function BacktrackPruning(state)

    if GoalReached(state) then
        return state
    end if

    state ← ForcedMovePropagation(state)
    if state = INVALID then
        return NULL
    end if

    color ← MRV(state)

    for each move in ManhattanOrderedMoves(state,
color) do

        next ← ExtendPath(state, color, move)

        if NOT PassesConstraintPruning(next) then
            continue
        end if

        solution ← BacktrackPruning(next)

        if solution ≠ NULL then
            return solution
        end if

    end for

    return NULL

end function
```

VI. Result and Discussion

```
Puzzle Level: 4

Facility Pairs:
  Red: 0,4 -> 1,2
  Blue: 1,3 -> 3,3
  Green: 2,3 -> 4,0
  Yellow: 4,1 -> 4,4

Expanded States: 9
Recursive Calls: 9
Backtracks: 0
Pruned States: 4
Execution Time: 511.6µs

=====================================
           SOLUTION BOARD
=====================================

   0   1   2   3   4
 0 o---o---o---o---R
   |
 1 o   o---R   B---o
   |   |           |
 2 o---o   o---G   o
           |       |
 3 o---o---o   B---o
   |
 4 G   Y---o---o---Y
```

Fig. 5. Program's Output. *(Source: Authors' Personal Archive)*

The experiment itself will be held across several test files that can be seen in the repository. This paper will discuss important findings only.

TABLE I. Several Instance Result Comparisons

File	Algorithm	Time (ms)	Nodes Expanded
3.json	BF	473	7975

	BT	66	8268
	BT+CP	1.996	53
5.json	BF	22488.365	7025
	BT	2.5	375
	BT+CP	1.003	13
invalid.json	BF	115219.652	1768
	BT	3.772	619
	BT+CP	1.092	11

Across all ten puzzle instances in the repository, the three algorithms consistently agreed on solvability. Every valid puzzle was solved in all five runs, while the invalid instance like invalid1.json was correctly rejected, demonstrating that constraint pruning preserves solution correctness. Performance, however, differed significantly. Brute Force was consistently the slowest algorithm, whereas both backtracking methods completed within a few milliseconds.

Comparing the two backtracking approaches, BT+CP substantially reduced the search space on difficult instances. For example, on 3.json, BT expanded 8,268 nodes compared to only 53 for BT+CP, reducing execution time from 66 ms to 1.996 ms. Similar improvements were observed on the infeasible puzzle, where BT+CP detected failure after expanding only 11 nodes versus 619 for BT. On simpler puzzles, however, BT+CP occasionally exhibited slightly higher execution times despite exploring fewer nodes, as the overhead of constraint checking outweighed its pruning benefits.

Overall, the results demonstrate that constraint pruning significantly improves search efficiency on complex and invalid puzzles while maintaining correctness. In contrast, Brute Force scales poorly because it exhaustively enumerates candidate combinations without performing early detection, resulting in substantially higher execution times as puzzle complexity increases.

VII. Conclusion

With this research, it can be concluded that state-space search algorithms provide an effective approach for solving the puzzle configurations found in the Pipe Work board game. This paper explored the reduction of the original tile-placement problem into a Flow Free-style representation, allowing the puzzle to be modeled as a graph-based search problem while preserving its connectivity constraints. Through the comparison of Brute Force, Plain Backtracking, and Backtracking with Constraint Pruning (BT+CP), it was shown that incorporating constraint-based pruning significantly reduces the search space and improves computational efficiency without sacrificing solution correctness, despite being NP-complete and may have exponential time complexity. This is handle well especially due to the nature of Pipe Work having fixed board grid.

Despite these results, there is still a lot of room for improvement. For instance, extending the proposed solver to support crossover pipe tiles, allowing it to solve the complete set of Pipe Work puzzle configurations without simplifying the original game mechanics. There are also other optimization techniques that might be available for further work. Some of these include using higher CSP methodology or even AI implementations that may guide the search or improve scalability. This area of work may come up with a better algorithm for the solutions with pipe abstractions as previously mentioned. Alas, author hopes this work can serve as a foundation for future research on efficient algorithms for solving Pipe Work and other similar combinatorial puzzle problems.

Youtube Video

https://youtu.be/yW-OMj3Vt98

Github Repository

https://github.com/jenka-h/pipework-boardgame

Acknowledgment

The author would like to express gratitude to everyone that supported her throughout the semester, despite the severity of situation. The author would also like to thank the lecturers of Algorithm Strategy IF2211, Mr. Rila Mandala, and the assistants for the vast knowledge they have shared throughout the course. The course itself was insightful and practical, the author hopes this may improve their studies in the Information Technology field. Lastly, the writer will forever be in debt to her parents. Without their support, the author would not have advanced this far in life.

References

[1] Rinaldi, Itb.ac.id, 2025. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/stima25-26.htm (accessed Jun. 17, 2026).

[2] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed. New Jersey: Pearson, 2010

[3] G. A. D. Banci, "Resolvendo Pipe Mania como Planejamento," Universidade de Brasília, 2022. Accessed: Jun. 18, 2026. [Online]. Available: https://bdm.unb.br/bitstream/10483/33938/1/2022_GuilhermeAntonioDeusdaraBanci.pdf

[4] D. Král', V. Majerech, SgallJ., T. Tichý, and G. Woeginger, "It is tough to be a plumber," Theoretical Computer Science, vol. 313, no. 3, pp. 473–484, Feb. 2004, doi: https://doi.org/10.1016/j.tcs.2002.12.002.

[5] "Flow Free solver," Needlessly complex, Aug. 28, 2016. https://mzucker.github.io/2016/08/28/flow-solver.html (accessed Jun. 19, 2026) .

STATEMENT

I hereby declare that this paper is my own original work. It is not a copy, translation, or adaptation of any other author's work, nor is it plagiarized.

Bandung, 1 Juni 2026

Jennifer Khang dan 13524110