Numberlink Final Write-Up

Arush Mehrotra & Tuneer Roy

Link to run project:
https://drive.google.com/file/d/1dNV396chlOeVxM2GeaU5mnq65Ax5dcuA/view?usp=sharing

Introduction

Inspiration

Before the project, we didn't know what Numberlink was but it was one of the recommended projects, and as we looked into it more, we thought it was quite interesting and would likely have interesting solutions. More so, beyond just being an interesting problem, it's also a puzzle that has been gamified and is available on app stores and gaming websites. It requires an interactive user interface that we thought would be fun to build and design. We thought it would also be interesting to try different solutions (different algorithms) with perhaps even different styles (ex: Constraint Programming solver vs. SAT solver) using the various libraries that we learned in class, and then benchmark and compare the results.

Functionality

Frontend (React)

- **Puzzle Generation Options**: Dropdown menus allow users to select puzzle difficulty levels (e.g., "Difficulty 5" corresponding to 5x5 puzzles) and choose from multiple generators (Generator 1, Generator 2, Generator 3). Once configured, users can click the "Generate Puzzle" button to create a unique puzzle on the grid below.
- **Solver Integration**: A solver panel enables users to select solvers ("PycoSAT Edge Solver", "PycoSAT Path Solver", "CP-SAT Path Solver") and difficulty levels for solving puzzles. The "Solve" button triggers the solver, filling in the grid.
- **Reset Functionality**: Users can reset the grid to a blank state, allowing for experimentation with different solvers.
- **Custom Puzzle:** Component to fill in own Numberlink puzzle (left-click to increment the key and right-click to decrement key) and run the solvers on them.

Backend (FastAPI)

- Supports all necessary endpoints for the frontend such as puzzle generation and solving the puzzles themselves using the various solvers.
- Also does slight optimizations to improve user experience where it will basically keep a buffer of puzzles at the ready for the user where the buffer is dynamically sized to accommodate demand. Specifically, if a user ever requests a puzzle and it's not already available, then the buffer increments in size.

Benchmarking

- **Performance Benchmarking**: We run benchmarks against the various solvers, measuring runtime and the number of clauses generated by the different solvers.

Generators

Initially, our project relied on what is now Generator 2. While functional, it proved to be extremely slow and inefficient for generating puzzles, particularly as the grid size increased. Additionally, the puzzles it created lacked sufficient complexity to pose meaningful challenges, often producing trivial configurations where numbers were directly adjacent.

To address these issues, we developed Generator 3. This new approach was significantly faster and improved the efficiency of puzzle generation. However, like Generator 2, it still failed to consistently generate interesting or difficult puzzles, as adjacent numbers frequently appeared, reducing the puzzle's challenge and compromising its usefulness for benchmarking solvers.

Ultimately, to properly benchmark and evaluate our solvers, we adopted Generator 1. This generator, which is from thomasahle/numberlink, was much faster and produced interesting, challenging puzzles. These qualities made Generator 1 a better choice for testing solver performance across a variety of puzzle difficulties and sizes.

Generator 1

This is from thomasahle/numberlink.

Generator 2

Generator 2 creates puzzles using a domino-based approach. It begins by constructing a grid where adjacent cells share the same value, representing dominos. The grid is randomized by swapping adjacent cells. Using a union-find algorithm, the generator ensures connections between dominos while avoiding cycles, with connectivity validated through depth-first search (DFS). The generator aims to minimize the number of dominos on the board by iterating up to 10,000 times to create simpler puzzles. After finalizing the solved version of the grid, a puzzle is generated by selectively removing information—zeroing out dominos that lack sufficient connectivity. The output is a tuple containing the unsolved puzzle and its solved version, ensuring the puzzle remains solvable and challenging.

Generator 3

Generator 3 takes a path-based approach to puzzle generation. It starts with an empty grid and iteratively adds paths by randomly selecting starting cells and connecting them to suitable neighbors. Each extension ensures that no isolated squares are created and that path constraints, such as avoiding excessive connections to a single path, are adhered to. The generator uses validation checks for neighboring cells and isolates to maintain the solvability of the puzzle. Once the board is fully covered with paths, it shuffles the path identifiers to randomize their positions, adding variety to the final puzzle. The result is a tuple with the unsolved and solved versions of the board, ensuring complete path coverage and adherence to constraints.

Solvers

To solve Numberlink, we experimented with three different solvers to investigate which was the most efficient. Broadly, we considered two different formulations: one in which we focused on the paths between pairs of numbers and one in which we focused on the edges between adjacent cells in the grid.[1] For the former formulation, we wrote two solvers: the ConstraintPathSolver and PycoPathSolver. Although these two solvers encode the same "constraints", we were curious if writing the solver as a set of constraints (a higher level of abstraction) or directly as a set of SAT clauses would result in better performance. Then, for the latter formulation, we wrote the PycoEdgeSolver. We wanted to compare which formulation was better for solving Numberlink puzzles quickly.

[1] https://www.lix.polytechnique.fr/~pilaud/enseignement/TP/DIX/INF421/1819/material/numberlink.pdf

ConstraintPathSolver

The ConstraintPathSolver encodes Numberlink as a list of constraints that must be satisfied. Consider a n x n grid, where we have j pairs of numbers that must be linked. Note that the longest path between each pair of numbers is (n^2). Then, it follows that we can define a boolean variable for each cell in the grid, for each potential path, and for each potential position on the path. Then, we can encode the constraints as follows:

- Each cell appears at a single position in a single path
- Each position in a path is occupied by one cell
- Consecutive cells along the path are adjacent to each other in the grid

We also must consider how to denote the "end" of a path since it intuitively follows that each path won't be n^2 long. To do so, we introduce an "end" cell that denotes that the path was finished before the current position p for a given path j.

Then, we have the last two constraints:

- Each position in a path is occupied by one cell (or the "end" cell if the path was finished before position p)
- The source (one of the cells with a number) appears first in a path and the sink (the other corresponding cell with the same number) is followed by "end" variables.

We utilize the CP-SAT Solver provided by OR-Tools.

Assuming we have n rows and n columns, if there are k numbers we must connect, the total number of variables becomes O(k·n^4).

For the constraints,

- **Vertex Occupancy Constraint**: Ensures that each vertex is visited at most once, requiring O(k·n^4) constraints for each pair and path position.
- **Position Occupancy Constraints**: Enforces that each position is visited at most once across all paths, requiring O(k·n^2) constraints.
- **Phantom Variable Constraints**: Enforces path finishing conditions, requiring O(k·n^2) constraints.
- **Path Continuity Constraints**: Ensures that each path step is continuous, requiring O(k·n^4) constraints.

- **Start and End Constraints**: Enforces source and sink conditions, requiring O(k·n^2) constraints.

PycoPathSolver

The PycoPathSolver translates the above constraints into an SAT problem and utilizes the PycoSAT library to solve it. Each variable is a representation of a cell (x, y), a path index (of the k possible paths), and a path position (1...n^2). We have an additional "phantom" variable that we use to represent when a path ends. For example, if a path is of size 4 but n = 3, then for path positions, 5 to 3^2, we use the "phantom vertex" for that particular path.

Across all clauses, we should have an upper bound of O(k^2 · n^6) clauses. If we assume k to be approximately n, which is typical, then it's O(n^8) clauses.

The number of clauses generated:

- **Vertex Occupancy Constraint:** each vertex is in a single path. So, for every possible vertex (of which there are n^2 possibilities), we want to enforce that there is exactly one true variable for each possible path position (of which there are n^2 possibilities) and for each possible path (of which there are k possibilities). So for each vertex, that is O((k·n^2)^2), or O(k^2·n^4). The outer squared term comes from the fact that "exactly one" among n literals requires O(n^2) clauses–refer to lecture notes for an explanation, it comes from the "at most one" bound. So in total, O(k^2·n^6) clauses across n^2 cells.
- **Position Occupancy Constraints**: each path position across all paths should have exactly one true variable for a cell or be the phantom vertex. Across all possible path positions (n^2), and across all paths (k), we have exactly one variable: O((n^2 + 1)^2) = O(n^4). So, across everything, O(k·n^6) clauses.
- **Phantom Variable Constraints**: if a path is finished at position x where x < n^2, it is finished at position x + 1. This is just 1 clause per possible path position and path: (NOT phantom at position x) AND (phantom at position x + 1). In all, O(k·n^2) clauses.
- **Path Continuity Constraints**: each vertex along a path must be neighbors. We add this constraint by ensuring that all vertices that aren't neighbors cannot be one position away from each other. In other words, an additional clause for each combination of vertices per path per path index, (NOT vertex v1 at path i at position x) AND (NOT vertex v2 at path i at position x + 1) for all vertices v1, v2 who aren't neighbors. This is in total O(k·n^6) clauses.

- **Start and End Constraints**: Finally, we need to enforce that the first path position is given to the path source and that the phantom variable follows the path sink. We do this by first having singleton clauses for all the path sources, i.e. (vertex sink_i at path i at position 0). We then enforce that for all path positions (n^2), if the sink is at position i, the phantom vertex follows at position i + 1. We also then enforce that the sink is on the path (exactly one) among n^2 literals. So, in total, O(k·n^2) clauses.

PycoEdgeSolver

The PycoEdgeSolver uses the PycoSAT library to solve the Numberlink problem in a different way. The SAT problem is formulated as follows: rather than considering each path between two numbers, we focus on each edge. Specifically, for a given path j, we consider if an edge e belongs to path j. Within the context of a grid, an edge corresponds to the neighbors of a cell. Then, we ensure that we satisfy the following constraints:

- Each edge is in AT MOST one path (doesn't have to be in any paths)
- Each vertex of a source/sink in a path is incident on exactly one edge.
- Each vertex NOT of a source/sink in a path is incident on exactly two edges.

Each variable is a representation of an edge on a given path (of which there are at most k). There are at most n^4 edges due to being at most n^2 cells.

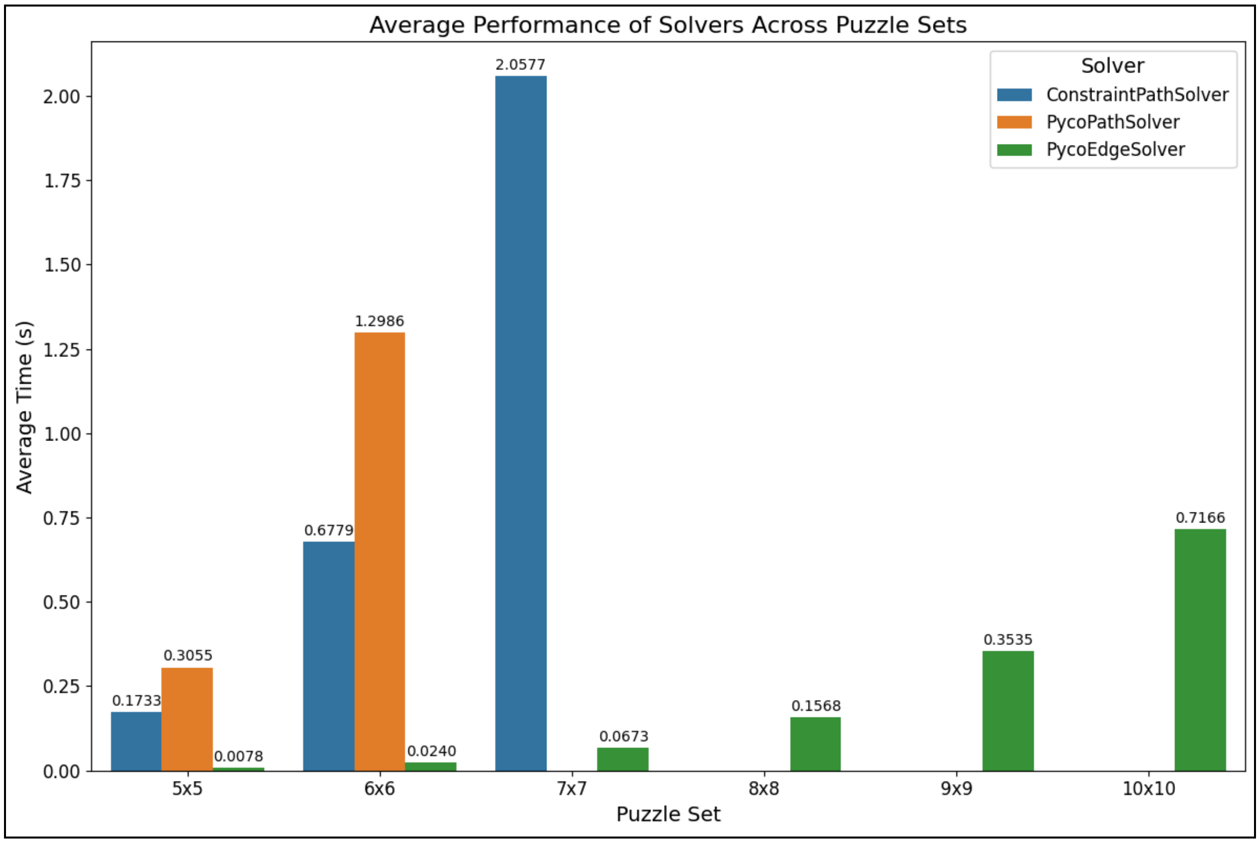
In total, the number of clauses is upper-bounded by O(k^3·n^2 + k^2·n^4) clauses. If we assume k to be approximately n, which is typical, then it's O(n^6).

The number of clauses generated:

- **Edge Constraint:** for each of the n^4 possible edges, we want at most one path (number of clauses generated for an "at most one" among n literals is O(n^2)), so O(k^2·n^4).
- **Path-Leaf Vertex Constraint:** we enforce that a source/sink of a path i is contingent on exactly one of the edges in path i, and also that they are contingent on zero edges in any other path. The former requires O(1) clauses per vertex per path index since each cell can have at most 4 neighbors. The latter requires O(k) clauses per vertex per path index. In total then, it takes O(k^2) since each path has at most 2 vertices that are sinks/sources.
- **Path-Inner Vertex Constraint:** this is the hard one since we have to enforce an "exactly two". There are O(n^2 - k) vertices who each need to be contingent on exactly 2 edges. We enforce this in a couple ways. First, we require that there be at least one edge for a given vertex across all paths, O(k) per vertex. Then we require that among any three edges, at most two are possible,

i.e. the clause (NOT x) AND (NOT y) AND (NOT z) where x, y, z are all possible contingent edges of a vertex across all possible paths. This is O(k^3) per vertex. Finally, we require that if an edge contingent on v is active in path i, we require another edge contingent on v to be active in path i, for any path-inner vertex v. This adds O(k) clauses. So, across all path-inner vertices, this is O((n^2 - k) · k^3) = O(k^3 · n^2 - k^4) = O(k^3·n^2).

Results



This bar chart compares the average performance times of three solvers—**ConstraintPathSolver**, **PycoPathSolver**, and **PycoEdgeSolver**—across puzzle sets of varying sizes (5x5, 6x6, 7x7, 8x8, 9x9, and 10x10). The following key points summarize the results:

1. **ConstraintPathSolver**:
   - Successfully completed puzzles for 5x5, 6x6, and 7x7 sets, with an increasing average time as the puzzles became larger (0.1733 seconds for 5x5, 0.6779 seconds for 6x6, and 2.0577 seconds for 7x7).
   - Did not complete puzzles for 8x8, 9x9, and 10x10 sets within the 10-second timeout.
2. **PycoPathSolver**:
   - Solved puzzles for 5x5 (0.3055 seconds) and 6x6 (1.2986 seconds) sets but did not solve larger puzzle sets within the timeout.
3. **PycoEdgeSolver**:

    - Demonstrated significantly faster performance for smaller puzzles, solving 5x5 in 0.0078 seconds and 6x6 in 0.0240 seconds.
    - Continued to solve larger puzzles (7x7 to 10x10) with increasing times, ranging from 0.0673 seconds for 7x7 to 0.7166 seconds for 10x10.

Observations:

- **Efficiency**: PycoEdgeSolver is the most efficient solver, capable of solving all puzzle sets up to 10x10, with its average times increasing steadily with puzzle size but remaining well below the timeout limit.
- **Scalability**: ConstraintPathSolver and PycoPathSolver struggle with larger puzzles, timing out beyond 7x7 and 6x6 respectively.
- **Performance Trends**: All solvers show increasing average times with puzzle size, reflecting the growing complexity of larger puzzles.

Conclusion/Analysis:

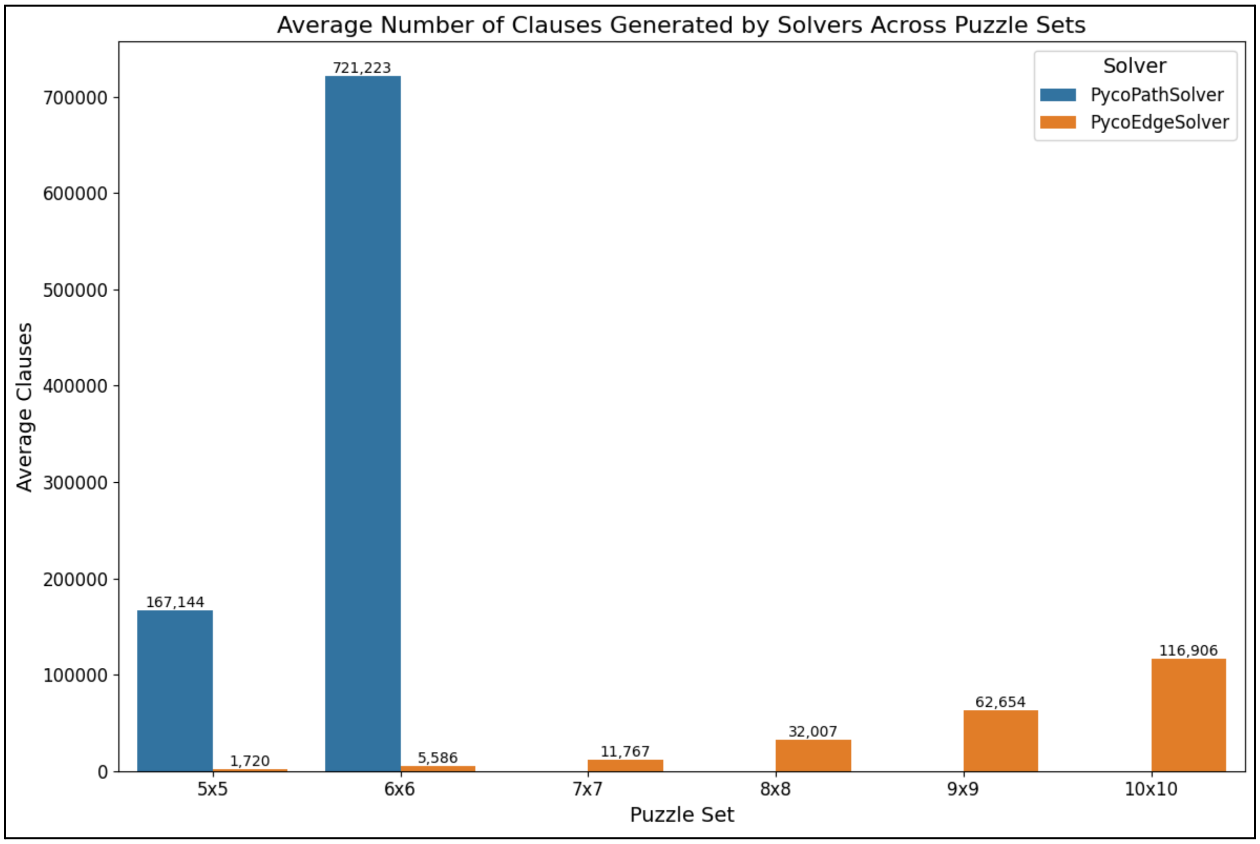
PycoPathSolver suffers from higher clause-generation complexity O(k^2 · n^6) because of the need to enforce "exactly one" constraints across path positions and vertices. This makes it less scalable, particularly for larger grids, which is reflected in its inability to solve puzzles larger than 6x6 within the set timeout of 10 seconds.

PycoEdgeSolver, with an upper bound of O(k^3·n^2 + k^2·n^4), is the most scalable among the PycoSAT approaches. It focuses on edges rather than paths, significantly reducing the number of variables and clauses while maintaining correctness. This makes it well-suited for cases with dense connections or a large number of paths.

The ConstraintPathSolver has an upper bound of O(k·n^4) in terms of the number of variables and constraints. However, it is important to note that it is not possible to extract accurate insights by comparing the asymptotic complexity of the CP-SAT and PycoSAT solvers. Although the CP-SAT solver translates the constraints into a SAT problem, from the user's perspective, we do not have insight into the number of SAT clauses that are generated. Given that the CP-SAT solver performs slightly better than the corresponding PycoPathSolver, it is likely that the internal translation into SAT clauses performed by the CP-SAT module is more efficient and optimized than our direct encoding of the path constraints using SAT clauses.

Given that the PycoEdgeSolver performed the best by far, it would be interesting to consider using the CP-SAT module to represent the edge constraints. We hypothesize that this approach would perhaps be the fastest solver due to the combination of a more efficient encoding strategy (focusing on edges) and the highly optimized internal processes of the CP-SAT solver. By representing edge constraints within the CP-SAT framework, we could potentially leverage the solver's advanced optimization techniques, such as lazy clause generation and efficient propagation mechanisms, to handle larger grid sizes and more complex puzzles.

In conclusion, PycoEdgeSolver is the most robust and scalable solver, consistently solving all puzzles within the time limit. ConstraintPathSolver and PycoPathSolver perform well for smaller puzzles but lack scalability for larger and more complex puzzles.



This bar chart displays the average number of clauses generated by two solvers—**PycoPathSolver** and **PycoEdgeSolver**—across different puzzle sets (5x5 to 10x10). Here are the key takeaways:

PycoPathSolver:

- Generates a significantly higher number of clauses compared to PycoEdgeSolver, indicating a computationally more intensive approach.
- The number of clauses increases dramatically with puzzle size:

    - **167,144 clauses for 5x5** puzzles.
    - **721,223 clauses for 6x6** puzzles.
- Does not show data for puzzle sets larger than 6x6 because it timed out.

PycoEdgeSolver:

- Generates far fewer clauses, showcasing a more efficient approach.
- The number of clauses grows steadily with puzzle size, reflecting its scalability:
    - **1,720 clauses for 5x5** puzzles.
    - **5,586 clauses for 6x6** puzzles.
    - **11,767 clauses for 7x7** puzzles.
    - **32,007 clauses for 8x8** puzzles.
    - **62,654 clauses for 9x9** puzzles.
    - **116,906 clauses for 10x10** puzzles.

Observations:

1. **Efficiency**: PycoEdgeSolver's efficiency is evident in its ability to handle significantly larger puzzles (up to 10x10) with a steady increase in clause generation.
2. **Scalability**: PycoEdgeSolver demonstrates excellent scalability, while PycoPathSolver struggles to handle larger puzzles, as evident by the lack of data for puzzle sets beyond 6x6.
3. **Clause Growth**: The rapid growth in clause generation for PycoPathSolver highlights potential inefficiencies in its algorithm compared to the more controlled growth in PycoEdgeSolver.

Conclusion:

PycoEdgeSolver outperforms PycoPathSolver in terms of clause generation efficiency and scalability. While PycoPathSolver creates exponentially more clauses, it fails to scale beyond 6x6, making PycoEdgeSolver the more reliable choice for larger and more complex puzzles.