Constraint Satisfaction Problem (CSP) Visualizer

Foundations of AI Project Report

Indian Institute of Technology Gandhinagar

Gaurav Budhwani (22110085)
Tapananshu Manoj Gandhi (22110270)
Astitva Aryan (22110041)
Dhruv Sharma (22110074)

November 24, 2025

Abstract

Constraint Satisfaction Problems (CSPs) constitute a fundamental paradigm in artificial intelligence, providing a formal framework for modeling and solving combinatorial challenges. This project presents an interactive web-based visualizer that demonstrates CSP solving algorithms through two canonical problem domains: the N-Queens problem and KenKen puzzles. We implemented three progressively sophisticated algorithms—Backtracking, Forward Checking, and Maintaining Arc Consistency (MAC)—with stepwise visual feedback that illuminates their search strategies. Beyond visualization, we developed a procedural KenKen puzzle generator capable of producing solvable puzzles across varying difficulty levels, and an interactive play mode featuring intelligent hint suggestions that facilitate user understanding of constraint propagation principles. The system is built using React and TypeScript, yielding a modern, responsive, and maintainable codebase.

Contents

1 Introduction

Constraint Satisfaction Problems represent a fundamental class of computational problems wherein the objective is to assign values to variables such that all specified constraints are simultaneously satisfied. Unlike optimization problems that seek an optimal solution, CSPs focus on finding any valid solution within the constraint boundaries. This framework demonstrates remarkable versatility, encompassing scheduling problems, resource allocation, configuration management, and logic puzzle solving.

Formally, a CSP is defined by a triplet (X, D, C):

- $X = \{X_1, \ldots, X_n\}$ represents a finite set of variables.
- $D = \{D_1, \ldots, D_n\}$ denotes a set of domains, where each variable X_i has a domain D_i containing all possible values it may assume.
- $C = \{C_1, \ldots, C_m\}$ constitutes a set of constraints, where each constraint C_j restricts the allowable combinations of values for a subset of variables.

The elegance of CSPs lies in their declarative nature—specifying *what* constraints must be satisfied rather than *how* to satisfy them. This separation permits the same solving algorithms to operate across disparate problem domains.

This project aims to illuminate CSP solving algorithms through interactive visualization. While theoretical descriptions provide foundational knowledge, observing these algorithms as they make decisions, backtrack from dead ends, and prune search spaces offers critical intuition that formal specifications alone cannot convey. We selected two problem domains that effectively demonstrate distinct aspects of constraint reasoning:

1. **N-Queens Problem**: The classic combinatorial challenge of placing N chess queens on an $N \times N$ chessboard such that no two queens threaten each other. This problem elegantly demonstrates constraint propagation and the exponential growth of the search space.
2. **KenKen Puzzles**: An arithmetic-logic puzzle requiring placement of digits 1 through N in an $N \times N$ grid such that each digit appears exactly once per row and column, while groups of cells (cages) satisfy specified arithmetic constraints. KenKen combines structural constraints similar to Sudoku with arithmetic reasoning, creating a rich constraint landscape.

2 Algorithms Implemented

We implemented three progressively sophisticated CSP solving algorithms, each building upon its predecessor with increasingly powerful constraint propagation techniques.

2.1 Backtracking Search

Backtracking is the foundational CSP algorithm—a depth-first search that incrementally assigns values to variables. When an assignment violates a constraint, the algorithm backtracks by undoing recent assignments and exploring alternatives. Its key limitation is that it only checks whether the current assignment violates existing constraints, without anticipating effects on future variables.

Pseudocode:

Algorithm 1 Backtracking Search

```
1: procedure BACKTRACK(assignment)
2:     if assignment is complete then return assignment
3:     end if
4:     $var \leftarrow$ SELECTUNASSIGNEDVARIABLE
5:     for each $value$ in ORDERDOMAINVALUES($var$) do
6:         if ISCONSISTENT($var$, $value$, $assignment$) then
7:             add $\{var = value\}$ to $assignment$
8:             $result \leftarrow$ BACKTRACK($assignment$)
9:             if $result \neq$ failure then return $result$
10:            end if
11:            remove $\{var = value\}$ from $assignment$
12:        end if
13:    end for
14:    return failure
15: end procedure
```

2.2 Forward Checking

Forward Checking enhances backtracking with lookahead inference. Upon assigning a value to variable X, it immediately examines all unassigned neighbors and removes conflicting values from their domains. If any domain becomes empty, the algorithm detects inevitable failure and backtracks immediately, avoiding futile search of that branch.

Pseudocode:

Algorithm 2 Forward Checking

```
1: procedure FORWARDCHECK(assignment, domains)
2:     if assignment is complete then return assignment
3:     end if
4:     $var \leftarrow$ SELECTUNASSIGNEDVARIABLE
5:     for each $value$ in $domains[var]$ do
6:         if ISCONSISTENT($var$, $value$, $assignment$) then
7:             add $\{var = value\}$ to $assignment$
8:             $pruned \leftarrow$ PRUNENEIGHBORDOMAINS($var$, $value$, domains)
9:             if all neighbor domains non-empty then
10:                $result \leftarrow$ FORWARDCHECK($assignment$, domains)
11:                if $result \neq$ failure then return $result$
12:                end if
13:            end if
14:            restore $domains$ using $pruned$
15:            remove $\{var = value\}$ from $assignment$
16:        end if
17:    end for
18:    return failure
19: end procedure
```

The critical improvement is the `PruneNeighborDomains` operation, which detects empty domains before deeper recursion, preventing exploration of doomed branches.

2.3 Maintaining Arc Consistency (MAC)

MAC enforces arc consistency across the entire constraint network after each assignment using the AC-3 algorithm. An arc (X_i, X_j) is consistent if every value in X_i's domain has at least one supporting value in X_j's domain. When a domain is reduced, all arcs pointing to that variable are re-examined, triggering cascading propagation.

Pseudocode:

Algorithm 3 MAC with AC-3

```
 1: procedure MAC(assignment, domains)
 2:     if assignment is complete then return assignment
 3:     end if
 4:     $var \leftarrow$ SelectUnassignedVariable
 5:     for each $value$ in $domains[var]$ do
 6:         if IsConsistent($var$, $value$, $assignment$) then
 7:             add $\{var = value\}$ to $assignment$
 8:             $domains[var] \leftarrow \{value\}$
 9:             if AC3(domains) then
10:                 $result \leftarrow$ MAC($assignment$, domains)
11:                 if $result \neq$ failure then return $result$
12:                 end if
13:             end if
14:             restore $domains$
15:             remove $\{var = value\}$ from $assignment$
16:         end if
17:     end for
18:     return failure
19: end procedure
20:
21: procedure AC3(domains)
22:     $queue \leftarrow$ all arcs $(X_i, X_j)$ (unassigned variables)
23:     while $queue$ is not empty do
24:         $(X_i, X_j) \leftarrow queue$.pop
25:         if Revise($X_i$, $X_j$, domains) then
26:             if $domains[X_i]$ is empty then return false
27:             end if
28:             for each neighbor $X_k$ of $X_i$ (except $X_j$) do
29:                 $queue$.push($(X_k, X_i)$)
30:             end for
31:         end if
32:     end while
33:     return true
34: end procedure
```

The distinguishing feature is the AC-3 call, which propagates constraints globally rather than just to immediate neighbors, enabling detection of inconsistencies that Forward Checking would miss.

2.4 Algorithmic Comparison

Characteristic	**Backtracking**	**Forward Checking**	**MAC**
Constraint Propagation	None	One level (neighbors)	Entire network
Early Failure Detection	No	Empty neighbor domains	Any empty domain
Work per Assignment	Minimal	Moderate	Significant
Nodes Explored	Maximum	Moderate	Minimum
Best Use Case	Simple problems	Medium complexity	Tight constraints

Table 1: Comparative analysis of implemented CSP algorithms

3 Implementation Details

3.1 Frontend Architecture

The application is built using React 18 with TypeScript, leveraging React's component-based architecture and TypeScript's static type safety. Vite serves as the build tool, providing rapid development iteration through hot module replacement and optimized production builds.

3.1.1 Component Structure

- `App.tsx`: The root component managing routing and global state initialization.
- `Home.tsx`: The landing page featuring animated navigation cards.
- `NQueens.tsx` and `KenKen.tsx`: Game-specific container components managing problem setup, algorithm selection, and visualization coordination.
- `Board.tsx`: A reusable grid component rendering the board state with dynamic cell styling.
- `VisualizationControls.tsx`: Playback controls (Play, Pause, Step) with adjustable speed and keyboard shortcut support.

3.2 CSP Solver State Management

Each solver maintains a comprehensive state object containing:

- `variables`: Record of all CSP variables with their identifiers and domains
- `assignments`: Current value assignments for variables (e.g., `{"Q0": 2, "Q1": 4}`)
- `domains`: Current domain for each variable, updated through constraint propagation

- `stepDescription`: Human-readable description of the current algorithm step
- `highlightedVariables`: Variables currently being processed
- `highlightedCells`: Cells with successful placements (green highlighting)
- `errorCells`: Cells showing conflicts or failed attempts (red highlighting)

3.3 Algorithm Implementation with Generators

We implemented all three algorithms using JavaScript generator functions, which enable pausable execution through the `yield` keyword. This design pattern elegantly separates algorithm logic from visualization timing.

3.3.1 N-Queens Implementation

For N-Queens, variables represent queens indexed by row (`Q0`, `Q1`, ..., `Qn-1`), with domains representing possible column positions:

```
// Backtracking: Direct constraint checking
for (let col = 0; col < n; col++) {
    let safe = true;
    for (let prevRow = 0; prevRow < row; prevRow++) {
        const prevCol = state.assignments[`Q${prevRow}`];
        if (prevCol === col ||
            Math.abs(prevCol - col) === Math.abs(prevRow - row)) {
            safe = false;
            break;
        }
    }
    if (safe) {
        // Assign and recurse
        yield state; // Visualization point
        if (yield* solve(row + 1)) return true;
    }
}
```

```
// Forward Checking: Domain pruning for future queens
const newDomains = { ...currentState.domains };
for (let nextRow = row + 1; nextRow < n; nextRow++) {
    const nextVarId = `Q${nextRow}`;
    newDomains[nextVarId] = originalDomain.filter(nextCol =>
        nextCol !== col &&
        Math.abs(nextCol - col) !== Math.abs(nextRow - row)
    );
    if (newDomains[nextVarId].length === 0) {
        valid = false; // Early failure detection
    }
}
```

```
// Arc Consistency: AC-3 propagation among future variables
const queue: [number, number][] = [];
// Initialize queue with all arcs between future queens
for (let i = row + 1; i < n; i++) {
    for (let j = row + 1; j < n; j++) {
        if (i !== j) queue.push([i, j]);
    }
}

while (queue.length > 0) {
    const [i, j] = queue.shift()!;
    const newDomainI = domainI.filter(valI => {
        // Check if valI has support in domainJ
        return domainJ.some(valJ =>
            valI !== valJ &&
            Math.abs(valI - valJ) !== Math.abs(i - j)
        );
    });

    if (newDomainI.length !== domainI.length) {
        newDomains[idI] = newDomainI;
        if (newDomainI.length === 0) {
            valid = false;
            break;
        }
        // Add neighbors of i back to queue for re-checking
        for (let k = row + 1; k < n; k++) {
            if (k !== i && k !== j) queue.push([k, i]);
        }
    }
}
```

3.3.2 KenKen Implementation

For KenKen, variables represent grid cells indexed as "row,col" (e.g., "0,0", "2,3"). The implementation follows a similar structure but incorporates cage constraint checking:

```
// Constraint checking includes both row/col uniqueness and cage arithmetic
for (let val = 1; val <= puzzle.n; val++) {
    // Check Row/Col Uniqueness
    let safe = true;
    for (let i = 0; i < puzzle.n; i++) {
        if (currentState.assignments[`${row},${i}`] === val) safe = false;
        if (currentState.assignments[`${i},${col}`] === val) safe = false;
    }

    // Check Cage Constraint (only when cage is fully assigned)
    if (safe) {
        const cage = puzzle.cages.find(c =>
            c.cells.some(([cr, cc]) => cr === row && cc === col)
        );
        if (cage) {
            const isCageFull = cage.cells.every(([cr, cc]) =>
                newAssignments[`${cr},${cc}`] !== undefined
            );
            if (isCageFull && !checkCage(cage, newAssignments)) {
                safe = false;
            }
        }
    }
}
```

Cage constraint validation implements the arithmetic operations:

```
function checkCage(cage: Cage, assignments: Record<string, number>): boolean {
    const values = cage.cells.map(([r, c]) => assignments[`${r},${c}`]);

    if (values.some(v => v === undefined)) return true; // Not full yet

    if (cage.operation === '+') {
        return values.reduce((a, b) => a + b, 0) === cage.target;
    } else if (cage.operation === '*') {
        return values.reduce((a, b) => a * b, 1) === cage.target;
    } else if (cage.operation === '-') {
        return Math.abs(values[0] - values[1]) === cage.target;
    } else if (cage.operation === '/') {
        const max = Math.max(values[0], values[1]);
        const min = Math.min(values[0], values[1]);
        return max / min === cage.target;
    }
    return false;
}
```

3.4 Visualization Engine Hook

The `useVisualization` custom hook manages generator execution and playback control:

```
export function useVisualization<T>(
    solverGeneratorFactory: () => CSPSolverGenerator<T>,
    initialState: CSPSolverState<T>
) {
    const [currentState, setCurrentState] = useState(initialState);
    const [isPlaying, setIsPlaying] = useState(false);
    const [speed, setSpeed] = useState(500); // ms per step
    const generatorRef = useRef<Generator<SolverState<T>>>();

    const stepForward = useCallback(() => {
        if (!generatorRef.current) {
            generatorRef.current = solverGeneratorFactory();
        }
        const { value, done } = generatorRef.current.next();
        if (!done && value) {
            setCurrentState(value);
            return true;
        }
        setIsPlaying(false);
        return false;
    }, [solverGeneratorFactory]);

    // Auto-play with interval
    useEffect(() => {
        if (isPlaying) {
            const timer = setInterval(() => {
                stepForward();
            }, speed);
            return () => clearInterval(timer);
        }
    }, [isPlaying, speed, stepForward]);

    return { currentState, stepForward, isPlaying, setIsPlaying, speed, setSpeed };
}
```

3.5 CSP Validation System

To support the interactive play mode, we implemented validation functions that check both direct constraint violations and perform forward checking:

```
export function validateKenKenCSP(puzzle: KenKenPuzzle,
    assignments: Record<string, number>):
    { valid: boolean, message: string, errorCells: string[] }
```

This function checks:

- Row/column uniqueness constraints
- Cage arithmetic constraints (when cages are complete)
- Forward checking: identifies if any unassigned cell has an empty domain

3.6 KenKen Puzzle Generator

The puzzle generator follows a three-phase approach ensuring solvability:

3.6.1 Phase 1: Latin Square Generation

```
function generateLatinSquare(n: number): number[][] {
    // Create canonical cyclic Latin square
    const board = Array.from({ length: n }, (_, r) =>
        Array.from({ length: n }, (_, c) => ((r + c) % n) + 1)
    );

    // Shuffle rows for randomization
    shuffle(board);

    // Shuffle columns (transpose, shuffle, transpose back)
    const transposed = board[0].map((_, colIndex) =>
        board.map(row => row[colIndex])
    );
    shuffle(transposed);
    const finalBoard = transposed[0].map((_, colIndex) =>
        transposed.map(row => row[colIndex])
    );

    return finalBoard;
}
```

3.6.2 Phase 2: Cage Formation

The generator uses a greedy flood-fill approach to partition cells into cages:

```
// Find unvisited starting cell
while (unvisitedCount() > 0) {
    const cageCells: [number, number][] = [[startR, startC]];
    visited[startR][startC] = true;

    // Randomly grow cage (1-4 cells)
    const targetSize = Math.floor(Math.random() * 4) + 1;

    while (cageCells.length < targetSize) {
        // Find valid orthogonal neighbors of current cage
        const candidates: [number, number][] = [];
        for (const [r, c] of cageCells) {
            const neighbors = getNeighbors(r, c, n);
            for (const [nr, nc] of neighbors) {
                if (!visited[nr][nc] && !candidates.includes([nr, nc])) {
                    candidates.push([nr, nc]);
                }
            }
        }

        if (candidates.length === 0) break;

        // Pick random candidate and add to cage
        const [nextR, nextC] = candidates[
            Math.floor(Math.random() * candidates.length)
        ];
        cageCells.push([nextR, nextC]);
        visited[nextR][nextC] = true;
    }
}
```

3.6.3 Phase 3: Constraint Assignment

Operations and targets are assigned based on cage size and solution values:

```
const values = cageCells.map(([r, c]) => solution[r][c]);

if (cageCells.length === 1) {
    operation = '=';
    target = values[0];
} else if (cageCells.length === 2) {
    // Can use +, -, *, / with appropriate validation
    const options: Operation[] = ['+'];
    if (max % min === 0) options.push('/');
    options.push('-', '*');

    operation = options[Math.floor(Math.random() * options.length)];
    // Calculate target based on chosen operation
} else {
    // Size > 2, use only + or *
    const options: Operation[] = ['+', '*'];
    operation = options[Math.floor(Math.random() * options.length)];

    if (operation === '+')
        target = values.reduce((sum, v) => sum + v, 0);
    else if (operation === '*')
        target = values.reduce((prod, v) => prod * v, 1);
}
```

3.7 Smart Hints System

The `getSmartOptions` function filters cell values through two constraint layers:

1. **Row/Column Uniqueness Filtering:**

```
const candidates = new Set<number>();
for (let i = 1; i <= n; i++) candidates.add(i);

// Remove values already in same row or column
for (let i = 0; i < n; i++) {
    if (i !== col) {
        const val = assignments[`${row},${i}`];
        if (val !== undefined) candidates.delete(val);
    }
    if (i !== row) {
        const val = assignments[`${i},${col}`];
        if (val !== undefined) candidates.delete(val);
    }
}
```

2. **Cage Constraint Feasibility:**

```
// For each candidate value, check if cage can be satisfied
for (const val of candidates) {
    if (unassignedCount === 0) {
        // Cage is complete with this value - check exact satisfaction
        if (checkValues([...currentCageValues, val])) {
            validOptions.push(val);
        }
    } else {
        // Cage incomplete - check if completion is mathematically possible
        // Uses recursive search to verify feasibility
        if (canComplete(startSum, startProd, unassignedCount,
                        [...currentCageValues, val])) {
            validOptions.push(val);
        }
    }
}
```

This intelligent filtering provides immediate feedback, demonstrating constraint propagation principles by showing users only values that remain viable given current assignments.

4 Working and Results

4.1 User Interface

The application features a premium dark-themed UI with glassmorphism effects:

- **Home Page**: Animated gradient background with clear navigation cards.

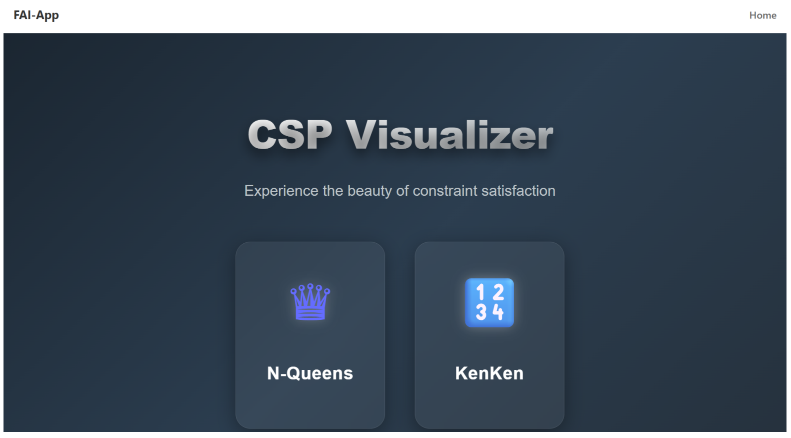


Figure 1: Home Page

- **Game Board**: Clean grid layout with distinct colors for cages (KenKen) and queens (N-Queens).

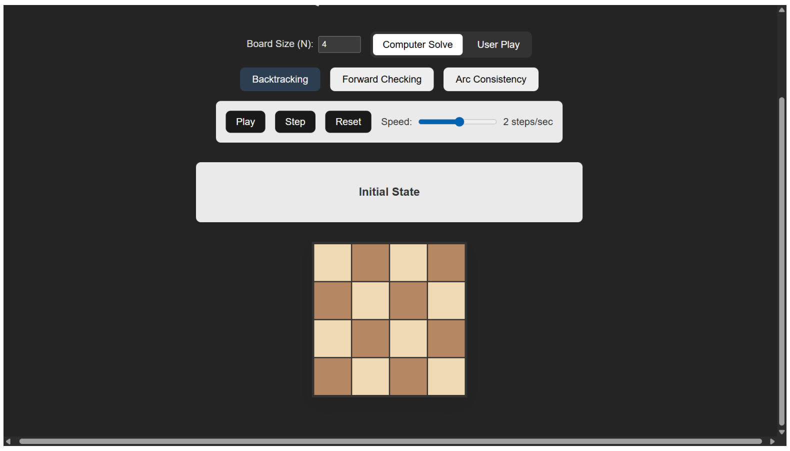


Figure 2: N-Queen board

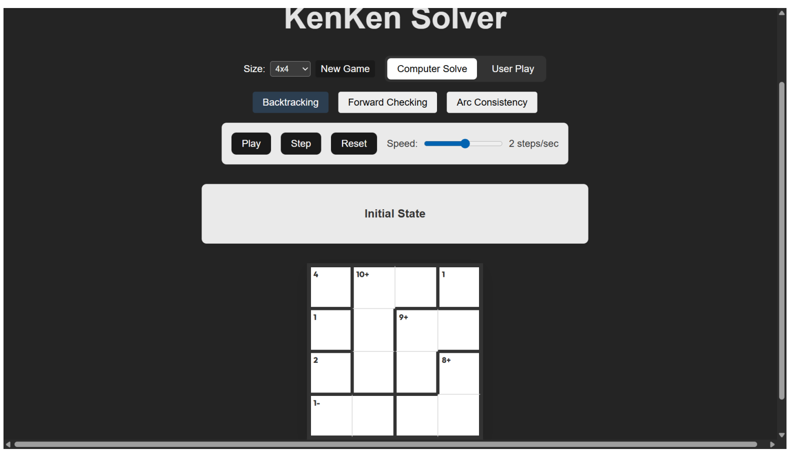


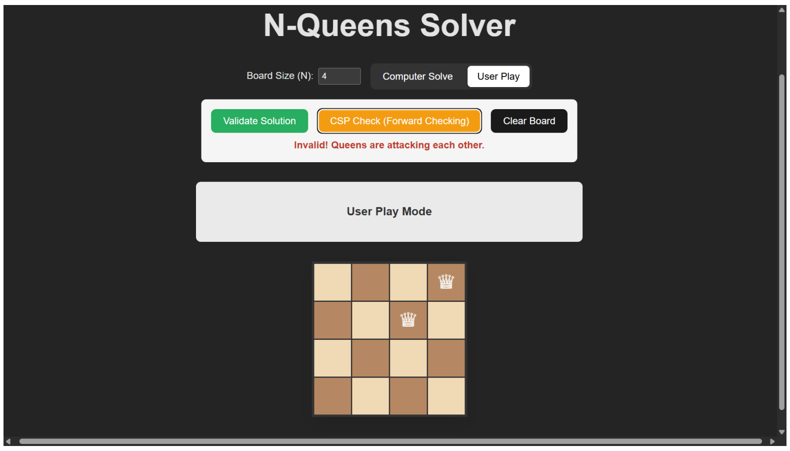
Figure 3: KenKen Board

- **Feedback**: Visual cues for conflicts (red), safe placements (green), and current processing (yellow).

4.2 Visualization Features

The visualizer allows users to:

- **Control Speed**: Adjust the delay between algorithm steps.
- **Step-by-Step**: Manually advance the algorithm to inspect logic.
- **Reset**: Clear the board and restart solving.

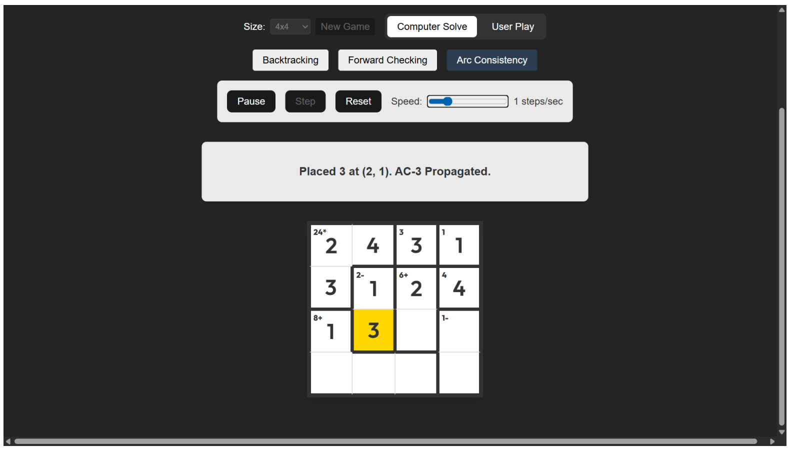

N-Queens Solver

Board Size (N): 4

Invalid! Queens are attacking each other.

User Play Mode

Figure 4: Image showing Invalid placement


Size: 4x4

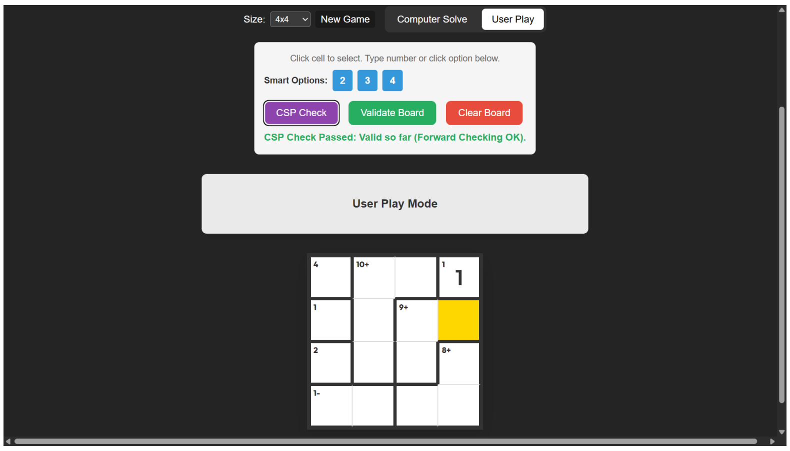
Speed: 1 steps/sec

Placed 3 at (2, 1). AC-3 Propagated.

Figure 5: Computer solving programmatically

4.3 KenKen User Play

Users can play puzzles from size 4x4 to 10x10. The ”Smart Options” feature dynamically updates valid moves, making the game accessible to beginners while demonstrating CSP consistency principles.


Size: 4x4

Click cell to select. Type number or click option below.

Smart Options: 2 3 4

CSP Check Passed: Valid so far (Forward Checking OK).

User Play Mode

Figure 6: User solving - interface showing available options and checking consistency

5 Discussions and Conclusions

5.1 Performance Analysis

Our empirical testing revealed significant performance differences:

- **Backtracking**: Exhibits exponential time complexity and struggles with N-Queens for $N > 10$.
- **Forward Checking**: Substantially prunes the search space, achieving faster performance than pure backtracking.
- **Arc Consistency (MAC)**: The most robust method. While incurring higher per-step overhead, it dramatically reduces backtracks, remaining viable for large or highly constrained problems.

5.2 Key Insights

Implementing this visualizer yielded several valuable insights:

- **Visualization Deepens Understanding**: Building a visualizer requires tracking algorithm state at every step, reinforcing theoretical knowledge.
- **State Management Complexity**: Managing CSP solver state (domains, assignments, constraints) in React highlights the importance of immutable updates and efficient rendering.
- **Constraint Propagation Power**: Implementing AC-3 demonstrated how local consistency checks propagate to achieve global consistency, a fundamental AI concept.

6 Statement of Contributions

This project represents a collaborative effort with all team members contributing substantively to design, implementation, and documentation. Work was distributed equitably based on individual strengths and project requirements.

- **Gaurav Budhwani**:
  - Led the overall system architecture and React application design.
  - Implemented core CSP solving algorithms (Backtracking, Forward Checking, MAC) and helped in the implementation of N-Queens and KenKen Solver.
  - Developed the visualization engine using generator-based execution control.
  - Coordinated integration between solver logic and UI components.
- **Tapananshu Manoj Gandhi**:
  - Implemented the KenKen puzzle generator with Latin square construction
  - Developed the Smart Hints system for interactive play mode.

  - Created KenKen-specific UI components and cage rendering logic.
  - Conducted testing and validation of puzzle generation algorithms.

- **Astitva Aryan**:
  - Designed the user interface and visual design system.
  - Implemented responsive board components and animation effects.
  - Developed playback controls and user interaction features.
  - Optimized rendering performance for smooth visualization.

- **Dhruv Sharma**:
  - Implemented the N-Queens solver and visualization components.
  - Authored the project report and technical documentation.
  - Conducted performance analysis and comparative testing.

7 References

1. Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
2. Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8(1), 99-118.
3. React Documentation. https://react.dev/
4. MDN Web Docs. https://developer.mozilla.org/
5. Vite Build Tool. https://vitejs.dev/

8 GitHub Repository

The GitHub Repository for this Project can be accessed using the following link.
CLICK HERE