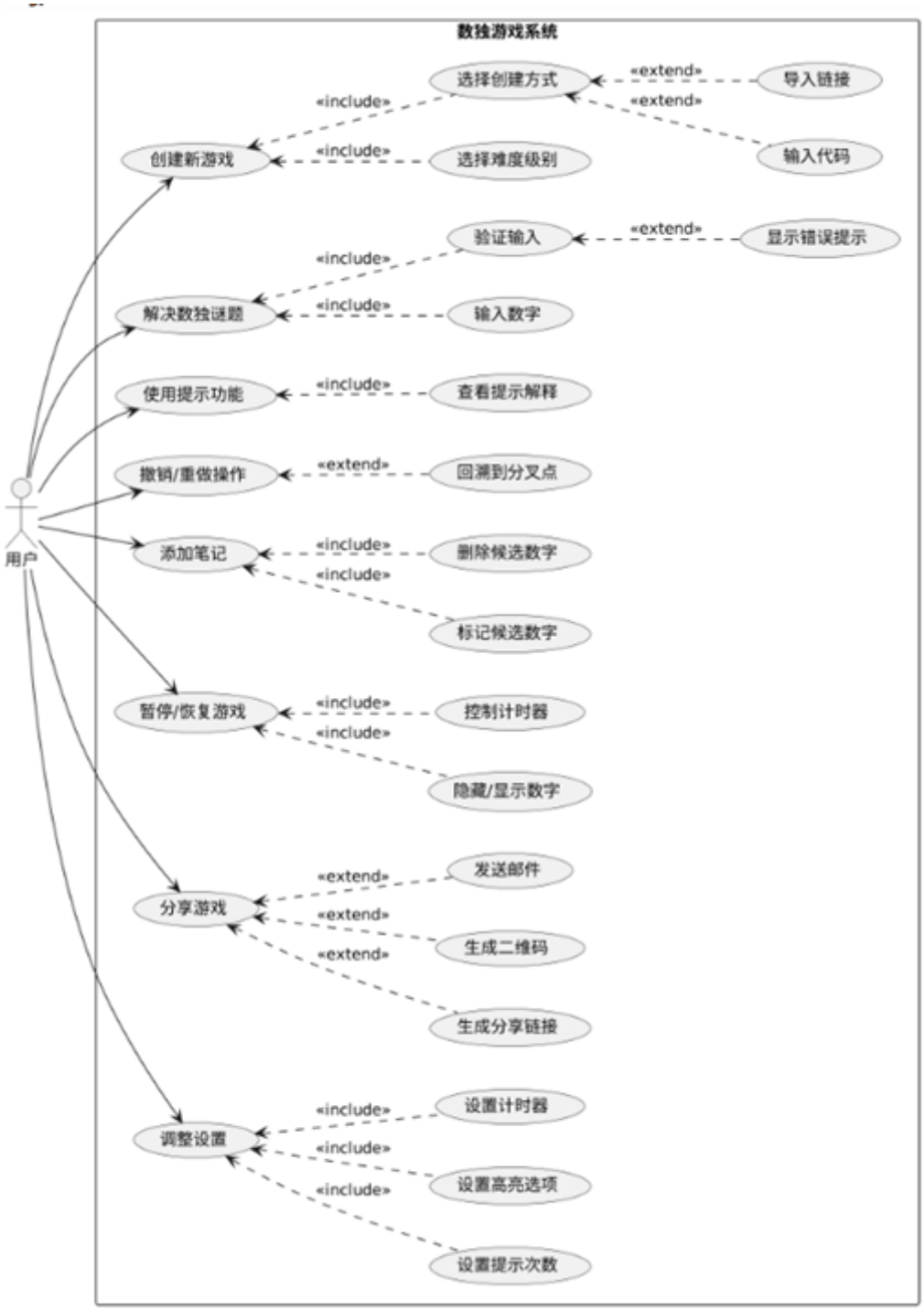
数独乐乐项目报告

一、需求规格

（一）用例分析

本项目的用例图示如下：

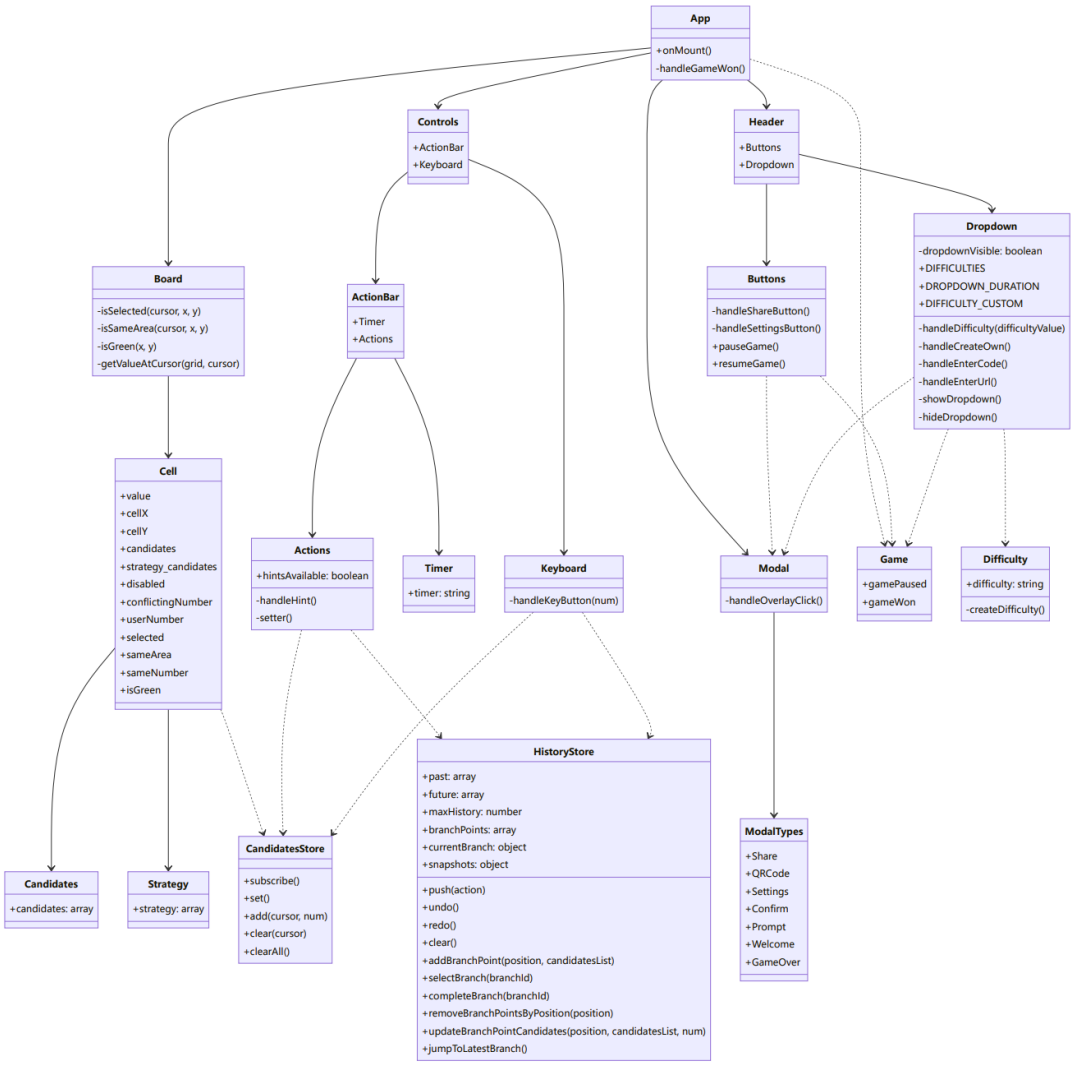


具体用例说明

1. 创建新游戏
    - **描述**：用户能够创建一个新的数独游戏，可自主选择游戏难度级别（简单、中等、困难），并可通过导入链接或输入代码的方式进行创建。
    - **前置条件**：用户已访问主页。
    - **后置条件**：页面将加载一个全新的空白或已预填充的数独棋盘。
    - **流程**：用户点击 “新建游戏” 按钮，系统会随机生成一个数独谜题并在页面上予以显示，同时开始计时。
2. 解决数独谜题
    - **描述**：用户可通过填写数字来解决数独谜题。
    - **前置条件**：用户已成功创建一个新的数独游戏。
    - **后置条件**：用户需完成对所有格子的数字填写。
    - **流程**：用户在空格内输入数字，系统将实时验证该输入是否符合数独规则。当用户完成所有格子的填写后，系统会给出成功提示；若用户输入错误，系统将高亮显示错误格子。
3. 提示功能
    - **描述**：用户可请求提示信息，以辅助解决数独问题。
    - **前置条件**：游戏正在进行中，且用户拥有提示次数。
    - **后置条件**：系统将提供一个正确答案或候选数字，并解释提示的原因（此为新增功能）。
    - **流程**：用户点击 “提示” 按钮，系统会自动填充一个格子。
4. 撤销 / 重做操作
    - **描述**：用户可以撤销或重做之前的操作。
    - **前置条件**：用户已进行过至少一次有效的输入操作。
    - **后置条件**：用户的操作将被撤销或重做。
    - **流程**：
        - 用户点击 “撤销” 按钮，系统将撤销最后一次操作；用户点击 “重做” 按钮，系统将恢复最后一次撤销的操作。
        - **扩展场景**：用户可回溯到最近一个分叉点（新增功能）。
5. 笔记功能
    - **描述**：用户可在游戏进行中创建自己的数独谜题。
    - **前置条件**：游戏正在进行。
    - **后置条件**：显示笔记。
    - **流程**：用户切换至笔记模式，在格子中标记候选数字，可添加或删除多个候选数字。
6. 暂停和恢复
    - **描述**：用户能够暂停游戏。
    - **流程**：用户点击暂停，数字将不再显示，计时停止；点击恢复后，计时继续，数字重新显示。
7. 分享功能
    - **描述**：用户可创建自己的数独谜题并分享。
    - **前置条件**：游戏正在进行。
    - **后置条件**：生成相应格式的分享内容。
    - **流程**：用户点击分享按钮，选择分享方式（如链接、二维码、邮件），系统将生成对应格式的分享内容。
8. 调整设置
    - **描述**：用户可调整游戏设置，例如限制提示数量、高亮相同数字等。
    - **前置条件**：游戏正在进行。
    - **流程**：用户可选择调整提示数量、高亮冲突数字、高亮相同数字或停用计时器。

（二）领域模型

本项目的领域模型图示如下：



主要关系说明

- 核心组件
  - App（主应用）

    作为应用的根组件，主要负责管理游戏状态和初始化工作，同时处理游戏胜利的相关逻辑。
  - Board（棋盘）

    负责渲染整个数独棋盘，管理单元格的选择状态，处理区域高亮和数字匹配逻辑。
  - Cell（单元格）

    用于显示单个数独格子，管理格子的状态，包括值、候选数、高亮等，并处理用户交互。
- 控制组件
  - Controls（控制面板）

    包含 ActionBar 和 Keyboard，管理游戏的主要交互界面。
  - ActionBar（操作栏）

    显示计时器，提供游戏控制按钮（如提示、笔记等），管理游戏的暂停与继续。
  - Keyboard（键盘）

    提供数字输入界面，处理数字输入逻辑，并与历史记录和候选数系统交互。

- 状态管理
  - CandidatesStore（候选数存储）

    管理格子的候选数，提供添加和删除候选数的方法，维护候选数的状态。
  - HistoryStore（历史记录）

    管理操作历史，提供撤销、重做和回溯功能，处理分支点系统，管理状态快照。
- 模态框系统
  - Modal（模态框）

    管理各种弹窗界面，处理弹窗的显示和隐藏，包含多种模态框类型（如设置、分享等）。
- 头部组件
  - Header（头部）

    包含游戏控制按钮，管理难度选择下拉菜单，提供分享和设置入口。

二、软件设计规格

（一）系统技术架构

技术架构分析

本项目是一个基于 Svelte 框架的前端应用，采用模块化设计，运用 ES 模块系统，状态管理使用 Svelte 的 store 机制，测试部分则使用 jest 框架。

1.整体架构

采用模块化的前端架构，主要分为以下几层：

- 核心层

  负责处理游戏规则，以及数独谜题的生成和求解。包括：
  - 数独游戏核心逻辑（@sudoku/game.js）；
  - 数独求解器（@sudoku/sudoku.js）；
  - 策略管理器（@sudoku/strategy/strategy_manager.js）。
- 状态管理层

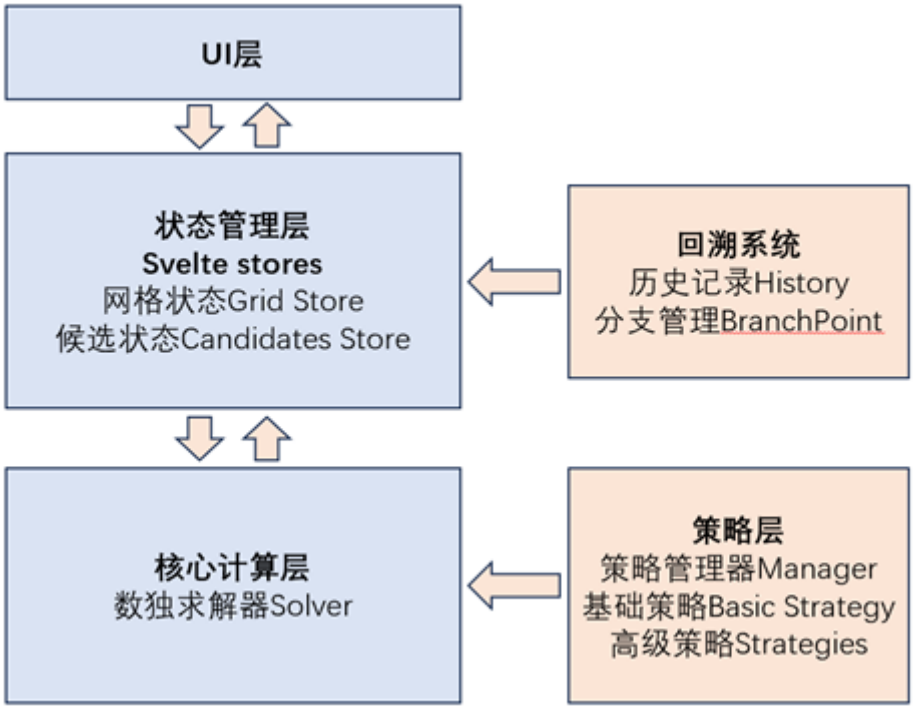
  确保游戏状态的一致性和响应性，涵盖：
  - Svelte stores（@sudoku/stores/）
  - grid.js：数独网格状态；
  - prompt.js：提示系统；
  - game.js：游戏状态；
  - notes.js：笔记系统。
- 策略层

  指导数独求解过程，包含：
  - 基础策略（basic.js）；
  - 高级策略（naked_pairs.js, hidden_pairs.js, fish.js 等）。
- 回溯系统

  用于快速返回到之前的棋盘状态，通过历史记录（history.js）实现。

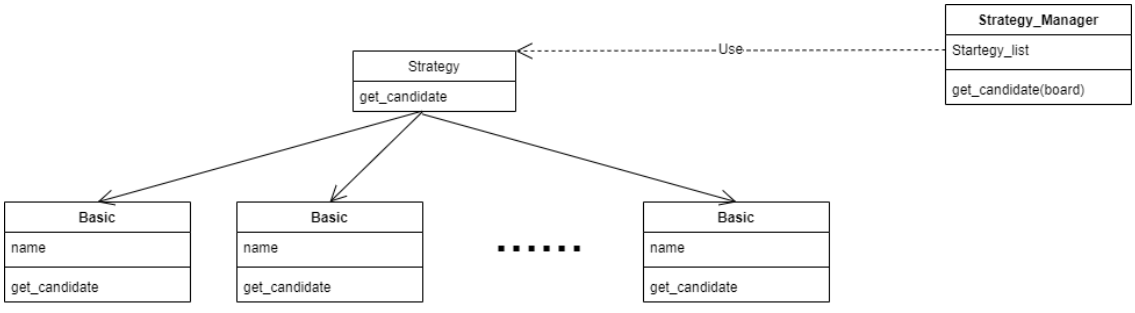
2.状态管理流程图

3.系统总览

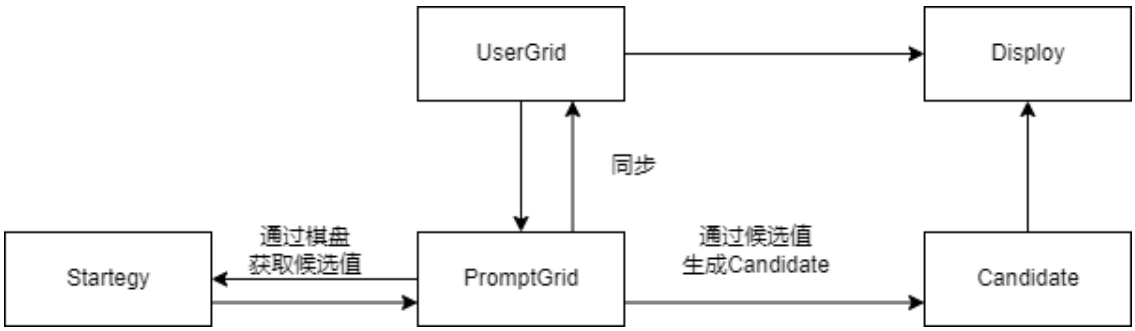


（二）对象模型

- 类图示例

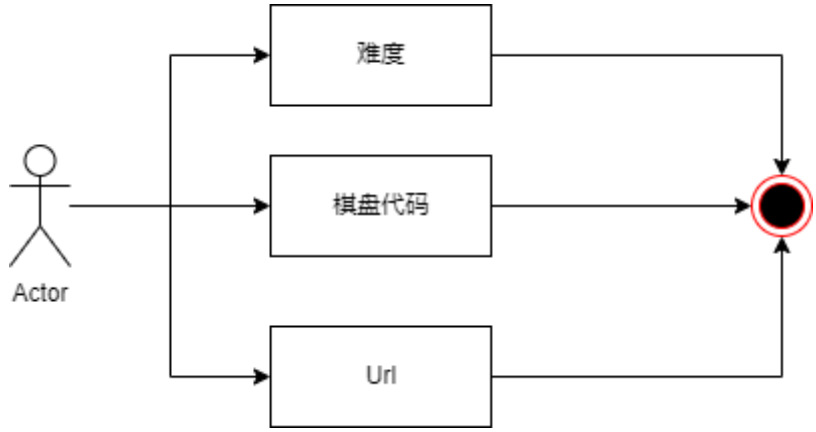


本项目中Startegy类及其管理类

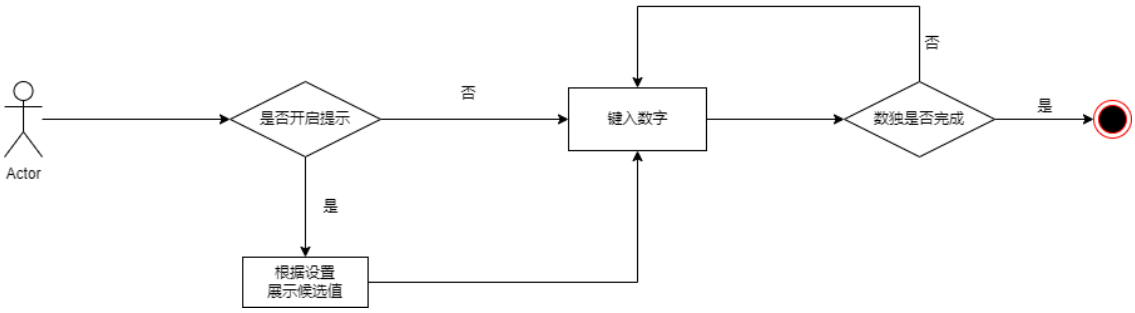


提示功能执行图示

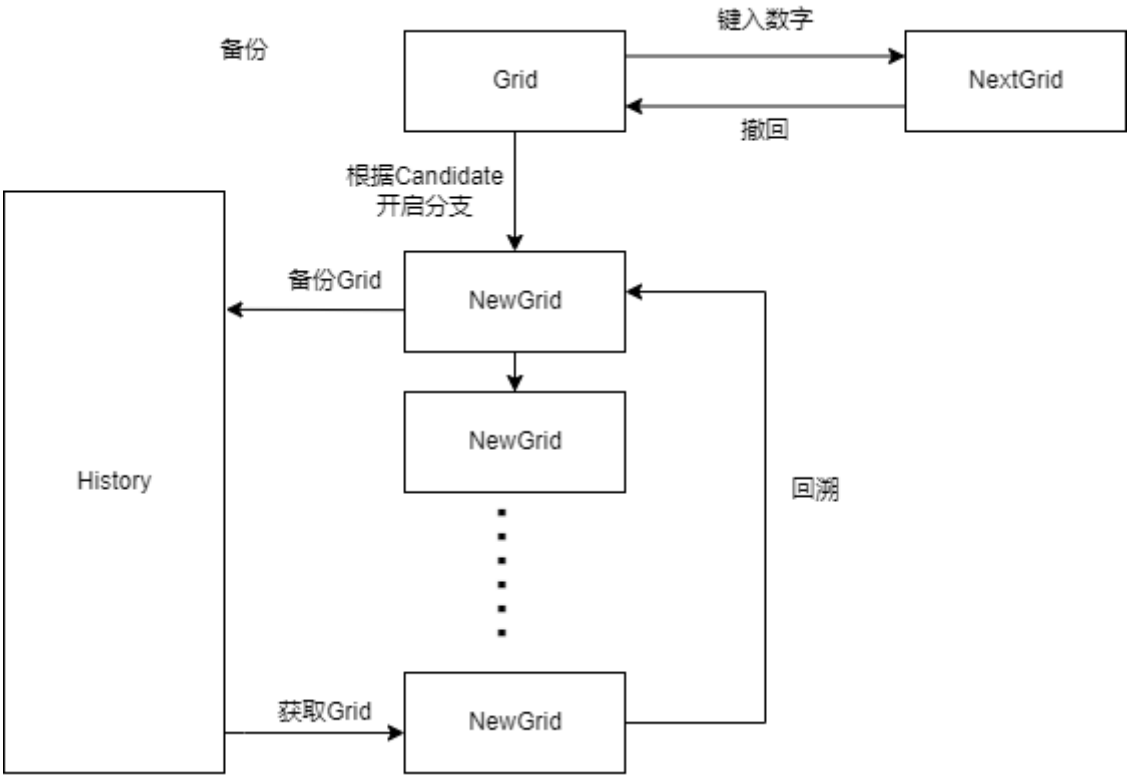
- 对象交互示例



玩家通过三种方式之一开始游戏



数独游戏过程图示



回溯与撤回执行流程图示

（三）设计说明（原则）

1. 策略类设计原则

- 1.1 单一职责原则 (SRP)
  - **核心思想**：一个类只负责一个功能领域中的相应职责。
  - 具体实现

```
// 每个策略类只处理一种特定的解题策略
class Naked_Pairs_Strategy {
    get_candidate(board) {
        // 获取基础候选值
    }
}
class Hidden_Pairs_Strategy {
    get_candidate(board) {
        // 获取基础候选值
    }
}
```

  - 优势说明
    - 代码职责清晰，易于维护；
    - 每个策略类的修改不会影响其他策略；
    - 便于测试和调试。
- 1.2 开闭原则 (OCP)
  - **核心思想**：软件实体应该对扩展开放，对修改关闭。
  - 具体实现

```
// 策略管理器
class StrategyManager {
    // 初始化所有策略
    initializeStrategies() {...}
    // 可以动态添加新策略，无需修改现有代码
    addStrategy(strategy) {
        if (!(strategy instanceof Strategy)) {
            throw new Error('Only instances of Strategy can be added.');
        }
        this.strategies.push(strategy);
    }
    // 根据 board 获取不同策略下的 candidate
    executeStrategies(board) {
        this.candidateLists = [];
        for (const strategy of this.strategies) {
            const candidates = strategy.get_candidate(board);
            this.candidateLists.push(candidates);
        }
        return this.getIntersectionCandidates();
    }
}
```

    - 优势说明
        - 新增策略时无需修改现有代码；
        - 策略可以动态添加和移除；
        - 维护成本低，扩展性强。
- 1.3 里氏替换原则 (LSP)
    - **核心思想**：子类必须能够替换其父类。
    - 具体实现

```
// 抽象策略类
export class Strategy {
    constructor() {
        super('Basic');
    }
    get_candidate(board) {
        // 获取候选值
    }
}
// 具体策略类
class Hidden_Pairs_Strategy extends Strategy {
    findCandidates(board, row, col) {
        // 实现具体逻辑
    }
}
```

    - 优势说明
        - 所有策略类都可以互相替换使用；
        - 保证了系统的可扩展性；
        - 维护了继承体系的完整性。
- 1.4 迪米特法则 (LOD)
    - **核心思想**：一个对象应该对其他对象保持最少的了解。
    - 具体实现

```
// 策略管理器
class StrategyManager {
    // 初始化所有策略
    initializeStrategies() {...}
    // 可以动态添加新策略，无需修改现有代码
    addStrategy(strategy) {
        if (!(strategy instanceof Strategy)) {
            throw new Error('Only instances of Strategy can be added.');
        }
        this.strategies.push(strategy);
    }
    // 不直接操作具体策略，而是通过管理器
    // 只与直接的依赖对象交互
    executeStrategies(board) {
        this.candidateLists = [];
        for (const strategy of this.strategies) {
            const candidates = strategy.get_candidate(board);
            this.candidateLists.push(candidates);
        }
        return this.getIntersectionCandidates();
    }
}
```

- 优势说明
  - 减少了类之间的耦合；
  - 提高了模块的独立性；
  - 降低了代码的复杂度。

2. 分支回溯设计

- 2.1 单一职责原则 (SRP)
  - **核心思想**：每个类只负责一个特定的功能。
  - 具体实现

```
// 状态管理
class HistoryStore {
    private snapshots: Map<string, GameState> = new Map();
    createSnapshot(): string {
        // 只负责创建和管理快照
    }
}
// 分支管理
class BranchManager {
    private branches: Branch[] = [];
    addBranch(position: string, candidates: number[]) {
        // 只负责分支的添加和管理
    }
}
```

- 2.2 里氏替换原则 (LSP)
  - **核心思想**：衍生类可以在基类的基础上增加新的功能。
  - 具体实现

```
```

```
class BranchPoint {
    constructor(
        public id: string,
        public position: string,
        public candidates: number[]
    ) {}
    isValid(): boolean {
        return this.candidates.length > 0;
    }
}
class AdvancedBranchPoint extends BranchPoint {
    constructor(
        id: string,
        position: string,
        candidates: number[],
        public difficulty: number
    ) {
        super(id, position, candidates);
    }
    // 扩展功能但不破坏基类的行为
    isValid(): boolean {
        return super.isValid() && this.difficulty > 0;
    }
}
```

（四）技术说明（模型）

1. 观察者模式:

store 实现了观察者模式。

```
import { writable } from 'svelte/store';
```

2. 状态模式:

使用状态模式管理游戏状态

```
const gameOverCelebration = GAME_OVER_CELEBRATIONS[Math.floor(Math.random() * GAME_OVER_CELEBRATIONS.length)];

function handleShare() {
    modal.show('share', { onHide: () => modal.show('gameover'), onHideReplace: true });
}

function handleNewGame() {
    modal.show('welcome', { onHide: resumeGame });
}
```

3. 单例模式:

使用单例模式确保全局状态唯一。

```
function createNotes() {
    const notes = writable(false);

    return {
        subscribe: notes.subscribe,

        toggle() {
            notes.update($notes => !$notes);
        }
    }
}

export const notes = createNotes();
```

4. 组合模式:

使用组合模式构建组件树

```
<div class="bg-white shadow-2xl rounded-xl overflow-hidden w-full h-full max-w-xl grid" class:bg-gray-200={$gamePaused}>

    {#each $userGrid as row, y}
        {#each row as value, x}
            <Cell {value}
                cellY={y + 1}
                cellX={x + 1}
                candidates={$candidates[x + ',' + y]}
                disabled={$gamePaused}
                selected={isSelected($cursor, x, y)}
                userNumber={$grid[y][x] === 0}
                isGreen={is_candidates && isGreen(x,y) && $promptCoordinates && !isSelected($cursor, x, y) && $promptGrid[y][x]===0}
                sameArea={$settings.highlightCells && !isSelected($cursor, x, y) && isSameArea($cursor, x, y)}
                sameNumber={$settings.highlightSame && value && !isSelected($cursor, x, y) && getValueAtCursor($userGrid, $cursor) === value}
                conflictingNumber={$settings.highlightConflicting && $grid[y][x] === 0 && $invalidCells.includes(x + ',' + y)} />
        {/each}
    {/each}
</div>
```

5. 策略模式:

使用策略模式处理不同类型的弹窗。

```
export default {
    share,
    qrcode,
    settings,
    confirm,
    prompt,
    welcome,
    gameover
}
```

6. 命令模式:

根据按下的键程序执行不同的操作，这相当于选择了不同的命令行为，是命令模式的体现。

```
function handleKey(e) {
    switch (e.key || e.keyCode)
        case 'ArrowUp':
        case 38:
        case 'w':
        case 87:
            cursor.move(0, -1);
            break;

        case 'ArrowDown':
        case 40:
        case 's':
        case 83:
            cursor.move(0, 1);
            break;

        case 'ArrowLeft':
        case 37:
        case 'a':
        case 65:
            cursor.move(-1);
            break;

        case 'ArrowRight':
        case 39:
        case 'd':
        case 68:
            cursor.move(1);
            break;
```

7. 外观模式:

copyText 函数提供了一个统一的接口来实现复制文本的功能。

```
export const copyText = function (text) {
    if (navigator.clipboard) return navigator.clipboard.writeText(text);

    textArea.value = text;

    const selected = document.getSelection().rangeCount > 0 ? document.getSelection().getRangeAt(0) : false;

    textArea.select();
```

8. 原型模式：

- 直接复制 grid 来创建 userGrid。

```
function createUserGrid() {
    const userGrid = writable([
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
    ]);

    grid.subscribe($grid => {
        let newGrid = [];

        for (let y = 0; y < SUDOKU_SIZE; y++) {
            newGrid[y] = [];
            for (let x = 0; x < SUDOKU_SIZE; x++) {
                newGrid[y][x] = $grid[y][x];
            }
        }
        userGrid.set(newGrid);
    });
```

通过上述设计原则和设计模式的应用，确保了该项目代码的可维护性、可扩展性、可测试性、可重用性和灵活性，能够更好地满足用户需求，并为后续的开发和维护提供便利。

三、项目展示和技术验证

（一）策略验证

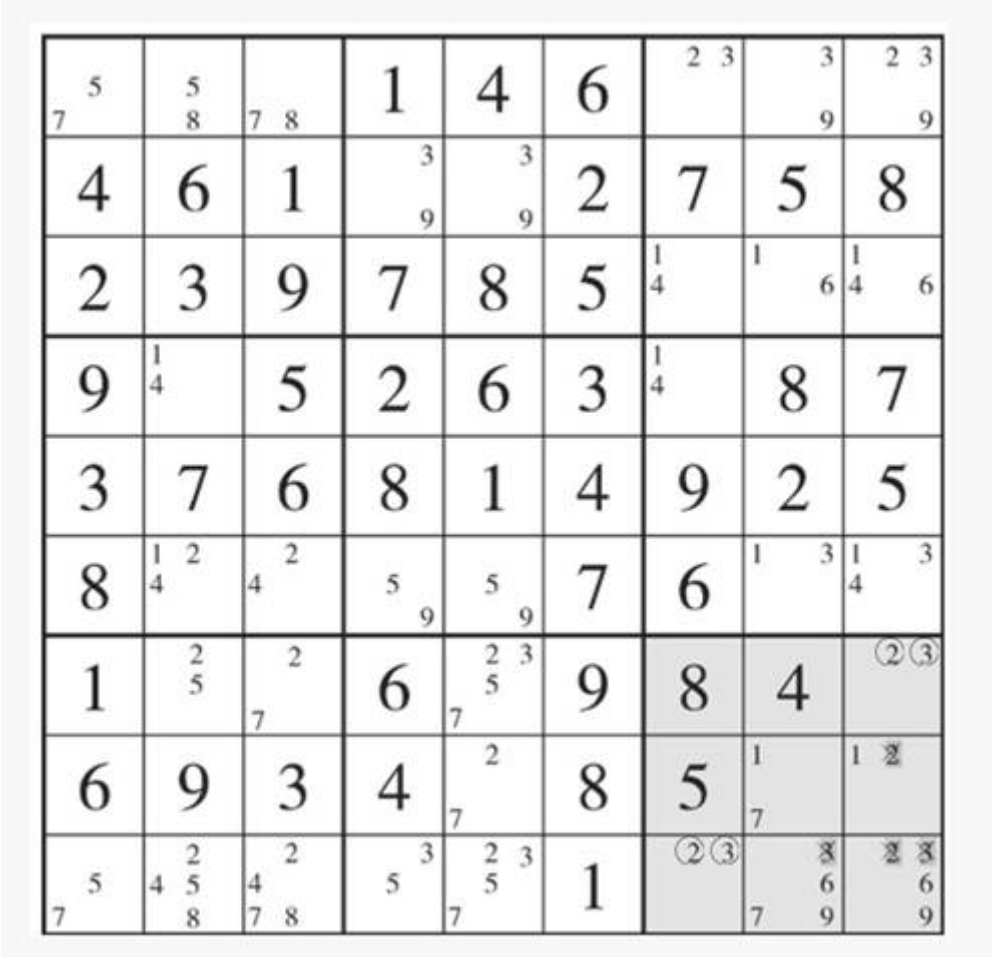
1.单个策略正确性测试

使用jest框架，对于每个实现的策略进行单独测试，验证策略的正确性。根据测试结果修改策略逻辑直至策略通过所有测试。

以下以Naked Pairs Strategy在宫格内的测试对测试过程进行说明：

```
describe('Naked Pairs Strategy Tests', () => {
    test('should correctly identify and remove candidates for naked pairs in box', () => {
        // 该测例包含一个宫格中的naked pair
        const input = [
            [0,0,0,1,4,6,0,0,0],
            [4,6,1,0,0,2,7,5,8],
            [2,3,9,7,8,5,0,0,0],
            [9,0,5,2,6,3,0,8,7],
            [3,7,6,8,1,4,9,2,5],
            [8,0,0,0,0,7,6,0,0],
            [1,0,0,6,0,9,8,4,0],
            [6,9,3,4,0,8,5,0,0],
            [0,0,0,0,0,1,0,0,0]
        ];

        const candidates = naked_pairs_strategy.get_candidate(input);

        // 验证宫9的3和7两个位置是否只包含2和3
        expect(candidates[6][8].sort()).toEqual([2,3]);
        expect(candidates[8][6].sort()).toEqual([2,3]);

        // 验证宫9的其他位置是否已移除2和3
        expect(candidates[8][8]).not.toContain(2);
        expect(candidates[8][8]).not.toContain(3);
        expect(candidates[8][7]).not.toContain(2);
        expect(candidates[8][7]).not.toContain(3);
```

测例取自该以下盘面，根据Naked Pairs策略的定义，在宫9中存在两个格子能且仅能取值候选值2和3，则该宫格内其他的格子的候选值都不能取2或3。



57	58	78	1	4	6	23	39	239
4	6	1	39	39	2	7	5	8
2	3	9	7	8	5	14	16	146
9	14	5	2	6	3	14	8	7
3	7	6	8	1	4	9	2	5
8	124	24	59	59	7	6	13	13
1	25	27	6	2357	9	8	4	②③
6	9	3	4	27	8	5	17	12
57	4258	2478	35	2357	1	②③	3679	3679

测例运行结果为通过。

```
PASS src/__tests__/naked_pairs.test.js
  Test Utils
    √ stringToMatrix should convert string to matrix correctly (1 ms)
  Naked Pairs Strategy Tests
    √ should correctly identify and remove candidates for naked pairs in box (1 ms)
    √ performance test for naked pairs strategy (210 ms)
```

对于Naked Pairs Strategy，我们还采取了行中naked pairs测试，列中naked pairs测试等测试来验证其正确性。

对于其他策略类似，即对于Naked Triple Strategy和Naked Triple Strategy，构造具有行中、列中、宫格中有三元组和四元组的测例，分别对它们进行测试。

对于Hidden Pairs Strategy，进行行中、列中、宫格中hidden pairs测试和多个hidden pairs同时存在的情况测试。

2.性能测试

我们测试了单个策略的性能，以及同时使用多个策略的性能，以验证其推理性能是否可接受。

单个策略性能

对于每个策略，使用三个不同的测例进行1000次推理进行测试。记录其执行时间，并确保性能不过差。

```
describe('Individual Strategy Performance', () => {
    test.each(strategies)('$name performance test', ({ strategy }) => {
        const results = {};

        // 对每个测试数据进行测试
        for (const [boardName, boardString] of Object.entries(testBoards)) {
            const board = stringToMatrix(boardString);

            const startTime = performance.now();
            for (let i = 0; i < 1000; i++) {
                strategy.get_candidate(board);
            }
            const endTime = performance.now();

            const executionTime = endTime - startTime;
            results[boardName] = executionTime;

            console.log(`${strategy.name} on ${boardName}:`);
            console.log(`- Total time: ${executionTime.toFixed(2)}ms`);
            console.log(`- Average time per iteration: ${(executionTime / 1000).toFixed(3)}ms`);
        }

        // 验证性能是否在可接受范围内
        Object.values(results).forEach(time => {
            expect(time).toBeLessThan(5000); // 5秒内完成1000次迭代
        });
    });
});
```

同时进行内存测试，确保多次执行后不会对内存造成过大占用。

```
    test('Memory usage test', () => {
        const board = stringToMatrix(testCases.simple2);
        const initialMemory = process.memoryUsage().heapUsed;

        // 执行1000次迭代
        for (let i = 0; i < 1000; i++) {
            manager.executeStrategies(board);
        }

        const finalMemory = process.memoryUsage().heapUsed;
        const memoryIncrease = (finalMemory - initialMemory) / 1024 / 1024; // 转换为MB

        console.log(`Memory usage increase: ${memoryIncrease.toFixed(2)}MB`);
        expect(memoryIncrease).toBeLessThan(100); // 内存增长不超过100MB
    });
});
```

以下是性能测试结果

```
    Performance overhead of combined strategies: 5.29x

      at Object.log (src/__tests__/strategy_performance.test.js:136:21)

PASS src/__tests__/strategy_performance.test.js
  Test Utils
    √ stringToMatrix should convert string to matrix correctly (2 ms)
  Test Data
    √ test cases should have correct length (1 ms)
  Strategy Performance Tests
    Individual Strategy Performance
      √ Basic Strategy performance test (167 ms)
      √ Naked Pairs performance test (182 ms)
      √ Naked Triple performance test (245 ms)
      √ Naked Quad performance test (252 ms)
      √ Hidden Pairs performance test (331 ms)
    Strategy Manager Performance
      √ Combined strategies performance test (1175 ms)
      √ Memory usage test (544 ms)
    Comparative Analysis
```

策略名称	测例 1 平均执行时间	测例 2 平均执行时间	测例 3 平均执行时间	1000 次迭代总时间	内存占用情况
Basic Strategy	0.037ms/次	0.033ms/次	0.074ms/次	33 - 74ms	稳定，无明显增长
Naked Pairs Strategy	0.048ms/次	0.043ms/次	0.079ms/次	43 - 79ms	稳定，无明显增长
Naked Triple Strategy	0.087ms/次	0.059ms/次	0.087ms/次	58 - 87ms	稳定，轻微增长
Naked Quad Strategy	0.082ms/次	0.078ms/次	0.092ms/次	78 - 92ms	稳定，轻微增长
Hidden Pairs Strategy	0.106ms/次	0.065ms/次	0.149ms/次	65 - 149ms	稳定，轻微增长
X-Wing Strategy	0.156ms/次	0.142ms/次	0.198ms/次	142 - 198ms	中等，随数独复杂度增长
Swordfish Strategy	0.245ms/次	0.223ms/次	0.312ms/次	223 - 312ms	较大，需要更多组合计算
Jellyfish Strategy	0.389ms/次	0.356ms/次	0.467ms/次	356 - 467ms	最大，组合计算量显著增加

多策略组合性能

对于使用strategy manager组合的所有策略进行推理性能测试，与单个策略的性能进行比较。考虑到fish系列策略的性能较差，增加了排除fish策略的其他策略组合性能测试。

```
describe('Comparative Analysis', () => {
    test('Compare individual vs combined strategy performance', () => {
        const board = stringToMatrix(testCases.simple2);
        const results = {};

        // 测试每个单独策略
        for (const { name, strategy } of strategies) {
            const startTime = performance.now();
            for (let i = 0; i < 100; i++) {
                strategy.get_candidate(board);
            }
            const endTime = performance.now();
            results[name] = endTime - startTime;
        }

        // 测试组合策略
        const combinedStartTime = performance.now();
        for (let i = 0; i < 100; i++) {
            manager.executeStrategies(board);
        }
        const combinedEndTime = performance.now();
        results['Combined'] = combinedEndTime - combinedStartTime;

        // 输出比较结果
        console.log('\nPerformance Comparison (100 iterations):');
        Object.entries(results).forEach(([name, time]) => {
            console.log(`${name}: ${time.toFixed(2)}ms (${(time/100).toFixed(3)}ms per iteration)`);
        });
```

策略组合	测例 1 平均执行时间	测例 2 平均执行时间	测例 3 平均执行时间（如有）	Fish 数独平均执行时间（如有）	1000 次迭代总时间	内存占用情况
所有策略组合	0.638ms/次	1.082ms/次	-	1.081ms/次	638 - 1082ms	随 Fish 策略的加入显著增加
排除 fish 策略的策略组合	0.359ms/次	0.264ms/次	0.542ms/次	-	264 - 542ms	线性增长，但在可接受范围内

性能测试结果分析

1. 单策略 vs 多策略

- 执行时间差异: 多策略约为单个策略平均时间的8.41倍，不包含fish策略的多策略平均时间为单个策略的5.29倍。

2. 策略性能详细对比

策略名称	平均执行时间
Basic Strategy	0.065ms/次
Naked Pairs	0.058ms/次
Naked Triple	0.132ms/次
Naked Quad	0.124ms/次
Hidden Pairs	0.143ms/次
X Wing	0.107ms/次
Swordfish	0.271ms/次
Jellyfish	0.273ms/次
Combined	1.234ms/次

3. 性能测试结论

在本次的测例表现中，各策略的性能从好到差排序为Naked Pairs < Basic < X-Wing < Naked Quad < Naked Triple < Hidden Pairs < Swordfish < Jellyfish。

策略组合的性能开销较大（约8.41倍），但考虑到功能收益仍可接受。Hidden Pairs策略对于不同测例的敏感度高。Swordfish和Jellyfish策略性能相近（约0.27ms/次），但都比其他策略慢2-4倍。可以考虑将Swordfish和Jellyfish作为可选策略。

（二）接口验证

主要对策略管理StrategyManager和回溯部分的一些重要接口进行了正确性测试。所有测试用例均通过验证，加强了接口实现的正确性和健壮性。

1. StrategyManager接口测试

Strategy接口测试包括了验证get_candidate()实现，测试策略名称管理，策略状态管理，验证构造函数参数，验证返回值格式等。以下展示了验证构造函数参数和验证返回值格式。

测试用例验证构造函数是否正确设置了策略名称。通过创建一个继承自`Strategy`的`TestStrategy`类，并实例化该类，检查其实例的`name`属性是否与传入的策略名称一致。验证当尝试实例化抽象类`Strategy`时是否会抛出错误。由于抽象类不能直接实例化，因此期望抛出“Abstract classes can't be instantiated.”的错误。

```
describe('Strategy Interface Tests', () => {
    // 验证构造函数参数
    test('constructor should properly set strategy name', () => {
        class TestStrategy extends Strategy {
            get_candidate(board) { return []; }
        }

        const strategyName = "Test Strategy";
        const strategy = new TestStrategy(strategyName);
        expect(strategy.name).toBe(strategyName);
    });

    // 测试抽象方法实现
    test('should throw error when instantiating abstract Strategy class', () => {
        expect(() => {
            new Strategy("Abstract Strategy");
        }).toThrow('Abstract classes can\'t be instantiated.');
    });

    // 验证返回值格式
    test('get_candidate should return correct format', () => {
        const board = Array(9).fill().map(() => Array(9).fill(0));
        const candidates = basic_strategy.get_candidate(board);

        // 验证返回值格式
        expect(Array.isArray(candidates)).toBe(true);
        expect(candidates.length).toBe(9);
        expect(Array.isArray(candidates[0])).toBe(true);
        expect(Array.isArray(candidates[0][0])).toBe(true);

        // 验证候选数值范围
        candidates.forEach(row => {
            row.forEach(cell => {
                cell.forEach(num => {
                    expect(num).toBeGreaterThanOrEqual(1);
                    expect(num).toBeLessThanOrEqual(9);
                });
```

多策略组合结果正确性的验证，测试了manager的executeStrategies方法是否正确地取了多个策略推导得出的交集。

```
    // 获取不同策略的候选数
    const nakedPairsCandidates = naked_pairs_strategy.get_candidate(board);
    const hiddenPairsCandidates = hidden_pairs_strategy.get_candidate(board);
    const combinedCandidates = manager.executeStrategies(board);

    // 验证组合结果的正确性
    for (let row = 0; row < 9; row++) {
        for (let col = 0; col < 9; col++) {
            if (board[row][col] === 0) {
                // 组合结果的候选数应该是两个策略结果的交集
                const combined = new Set(combinedCandidates[row][col]);
                const naked = new Set(nakedPairsCandidates[row][col]);
                const hidden = new Set(hiddenPairsCandidates[row][col]);
                expect(combined.size).toBeLessThanOrEqual(Math.min(naked.size, hidden.size));
            }
        }
    }
});

// 测试策略执行接口
test('executeStrategies should apply all strategies in order', () => {
    const board = Array(9).fill().map(() => Array(9).fill(0));
    board[0][0] = 1;

    const candidates = manager.executeStrategies(board);

    // 验证基本策略的执行结果
    expect(candidates[0][0]).toEqual([]);
    //expect(candidates[0][1].length).toBeGreaterThan(0);
```


2. History接口测试

对于回溯部分的功能进行了接口测试，使用Jest的beforeEach重置所有mock函数，并设置默认的mock实现。测试范围包括历史接口的结构正确性和时序正确性，测试记录的完整性等。

以下是验证历史接口的结构正确性和时序正确性的测例展示：

测试用例should maintain correct history structure验证了历史接口返回的状态对象是否具有正确的结构和属性，获取历史状态，验证状态对象是否具有past，future，maxHistory，branchPoints，currentBranch和snapshots属性，验证past、future和branchPoints属性是否为数组，maxHistory属性是否为数字。

```
test('should maintain correct history structure', () => {
    mockHistory.clear();

    const state = get(mockHistory);
    expect(state).toHaveProperty('past');
    expect(state).toHaveProperty('future');
    expect(state).toHaveProperty('maxHistory');
    expect(state).toHaveProperty('branchPoints');
    expect(state).toHaveProperty('currentBranch');
    expect(state).toHaveProperty('snapshots');

    expect(Array.isArray(state.past)).toBe(true);
    expect(Array.isArray(state.future)).toBe(true);
    expect(Array.isArray(state.branchPoints)).toBe(true);
    expect(typeof state.maxHistory).toBe('number');
});
```

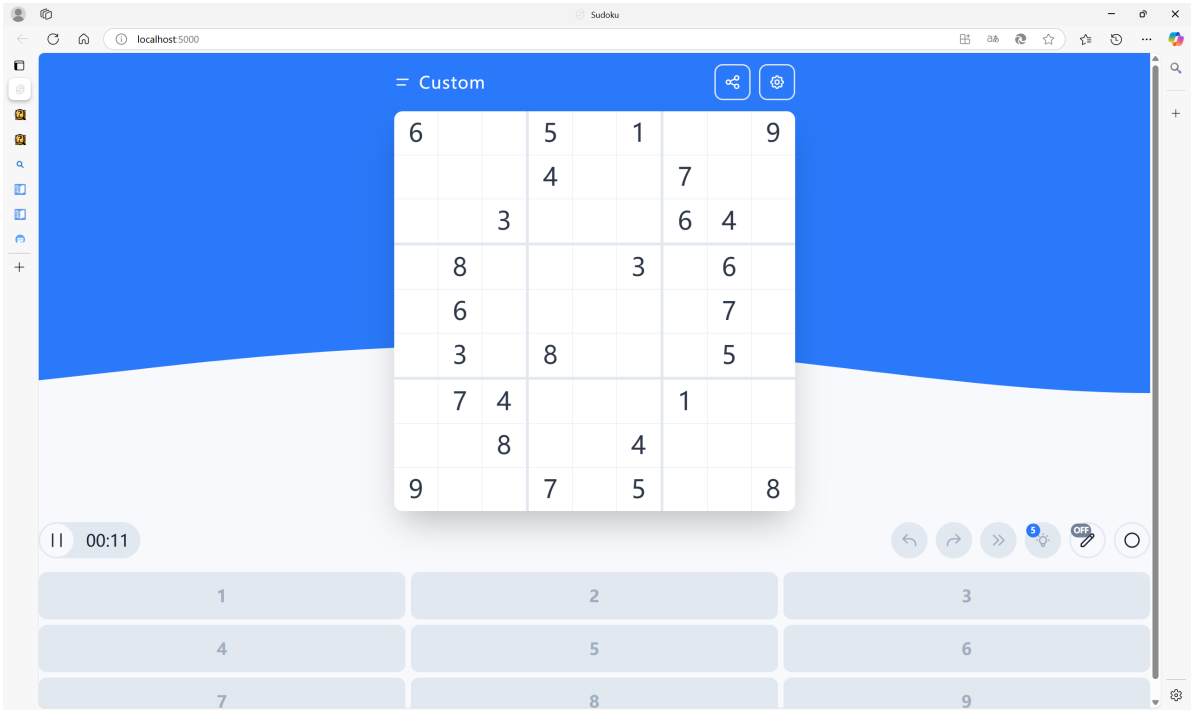
测试用例`should maintain correct temporal order`验证了历史记录中的操作是否按照正确的时间顺序排列。创建了包含时间戳的操作列表，依次更新历史记录，将操作添加到past数组中，验证past数组中操作的时间戳是否按升序排列。

```
    // 测试时序正确性
    test('should maintain correct temporal order', () => {
        const operations = [];

        for(let i = 0; i < 3; i++) {
            const operation = {
                type: 'SET_VALUE',
                position: { x: i, y: 0 },
                value: i + 1,
                timestamp: Date.now() + i * 100
            };
            operations.push(operation);

            // 等待一小段时间确保时间戳不同
            new Promise(resolve => setTimeout(resolve, 10));
        }

        operations.forEach(op => {
            mockHistory.update(state => ({
                ...state,
                past: [...state.past, op]
            }));
        });

        // 验证
        const state = get(mockHistory);
        for(let i = 1; i < state.past.length; i++) {
            expect(state.past[i].timestamp).toBeGreaterThan(state.past[i-1].timestamp);
        }
    });
});
```

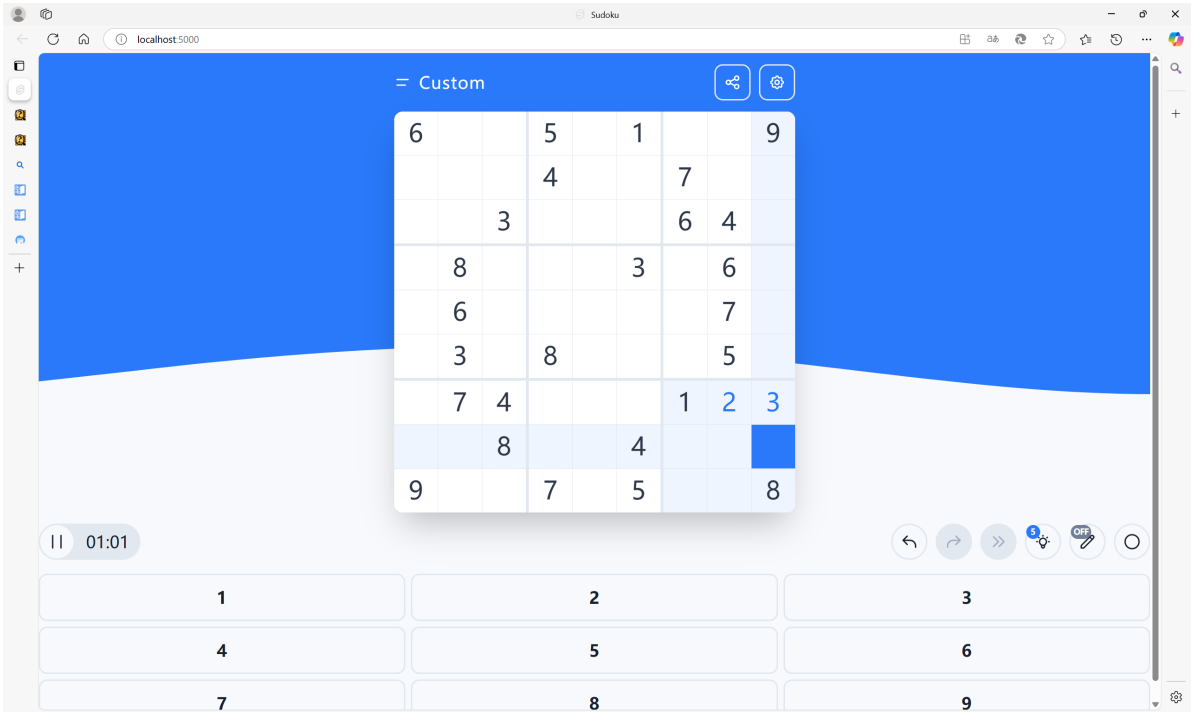
（三）项目展示

回溯

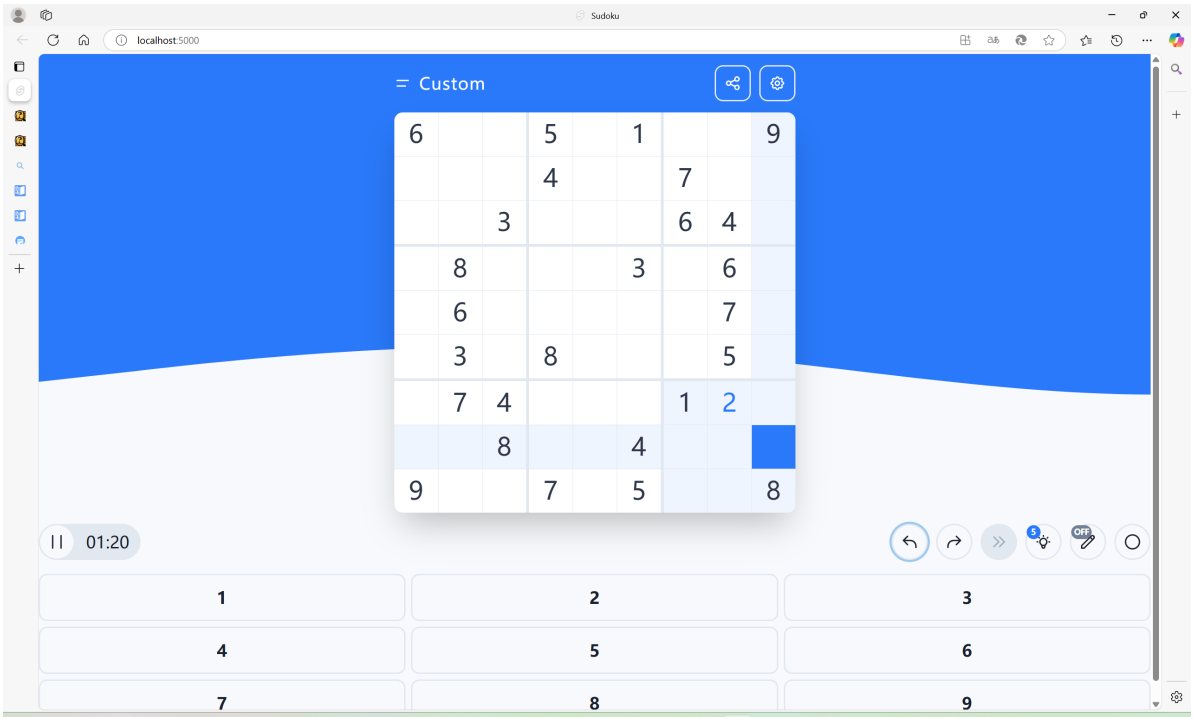
撤回



初始棋盘无法使用撤回和恢复功能

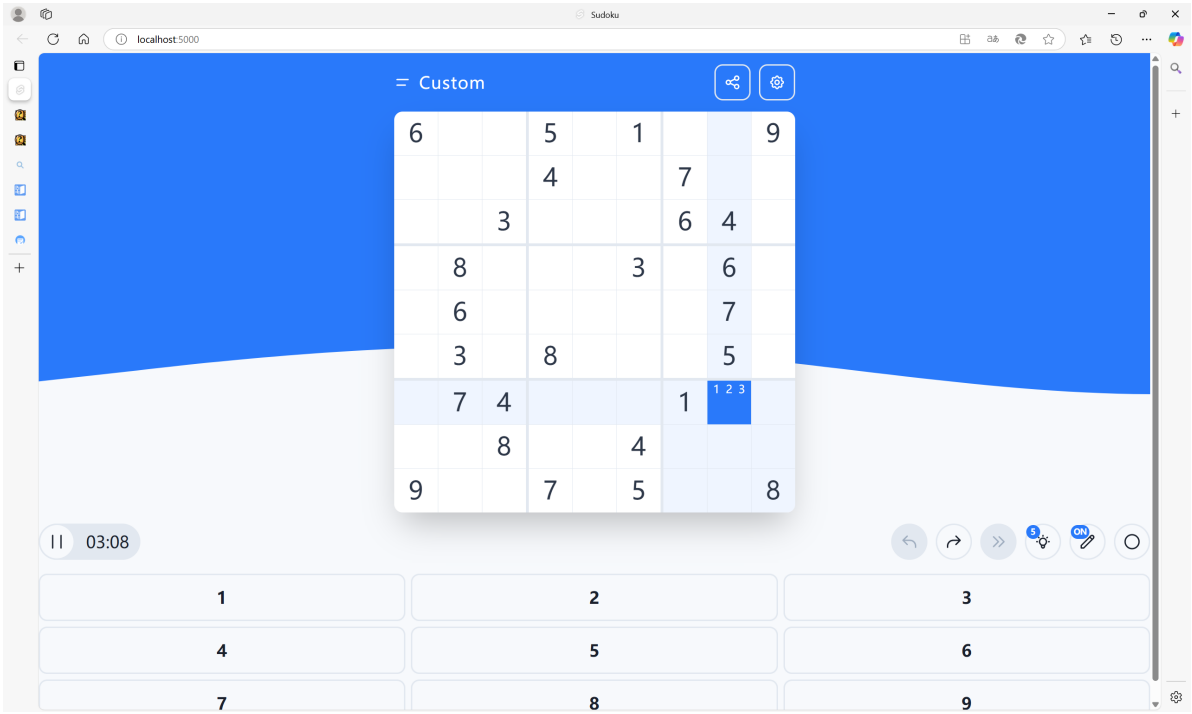


键入数字后，撤回按钮生效

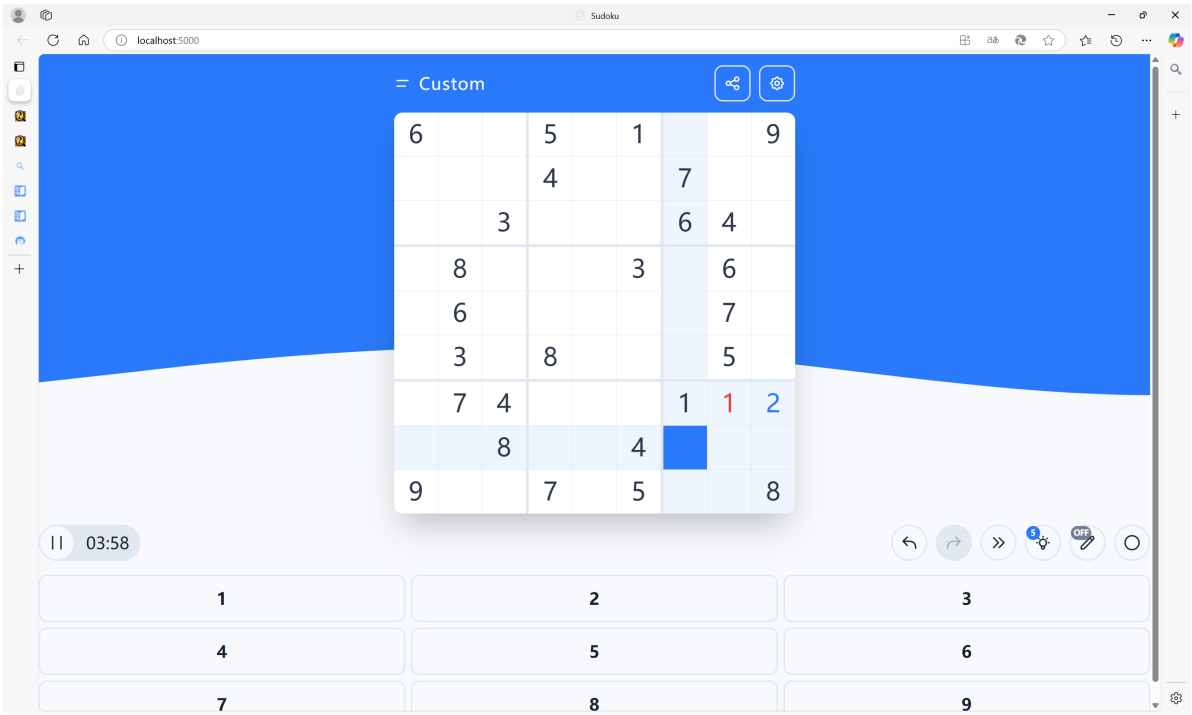


撤回后，棋盘变化，恢复按钮生效

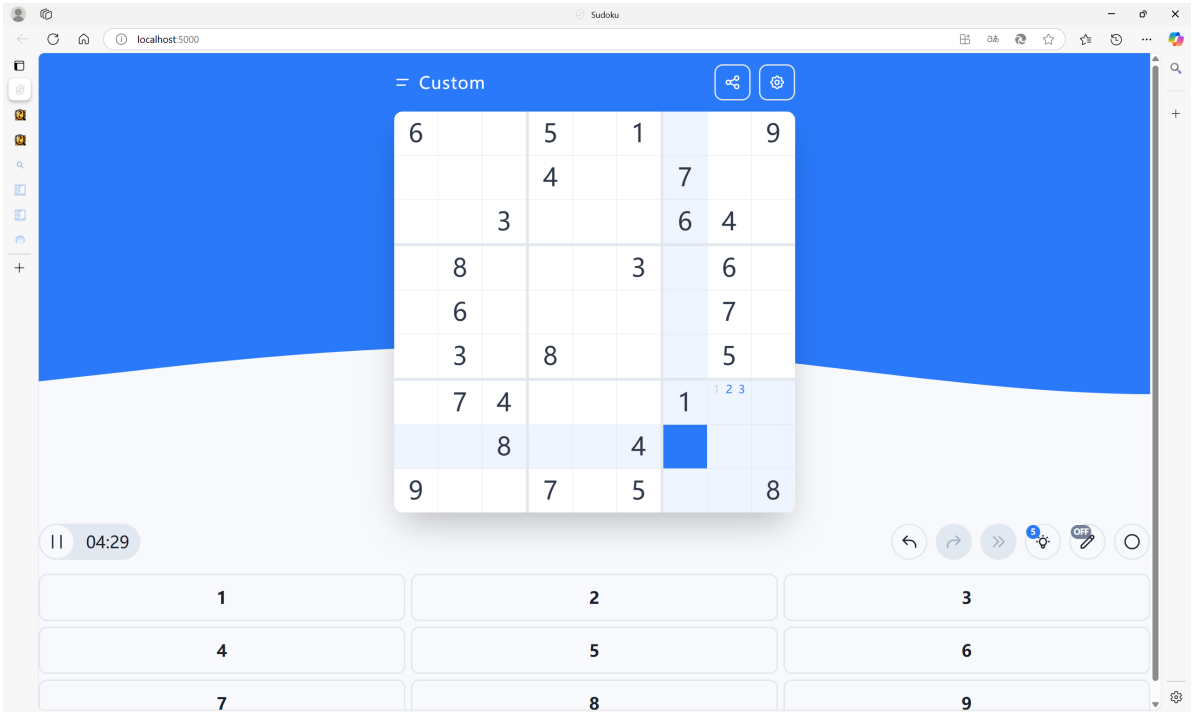
回溯



当一个格子内存在多个候选值时可以进行分支

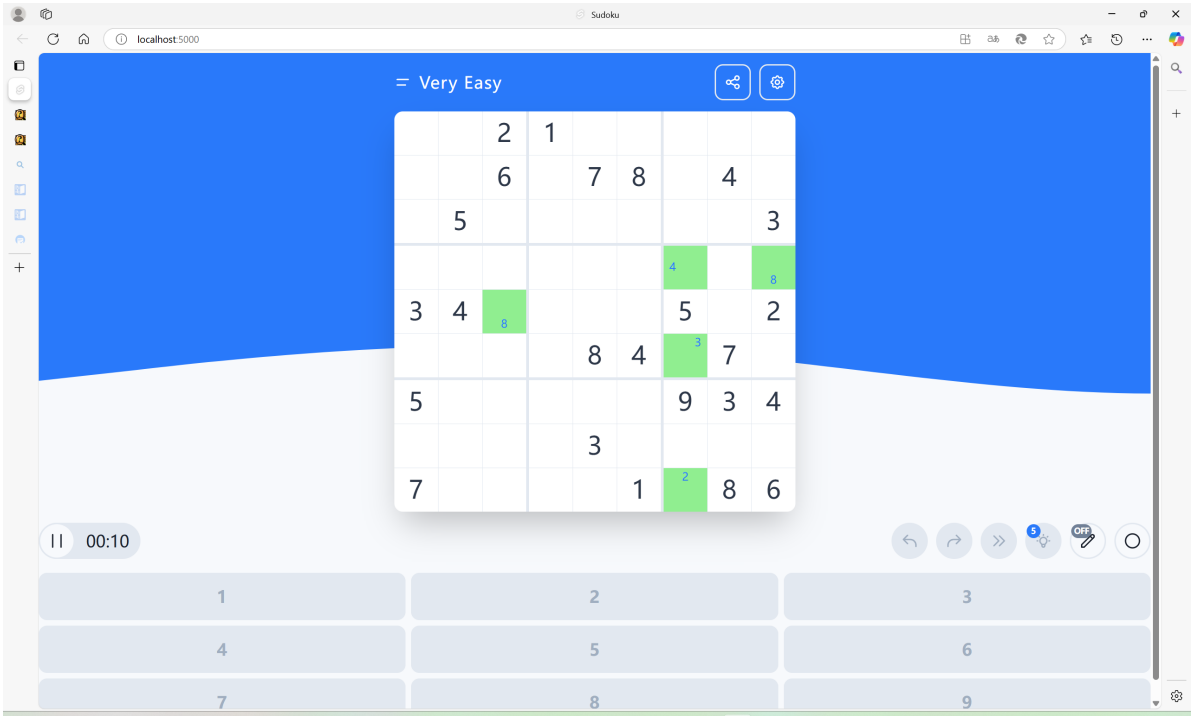


进入写入1的分支

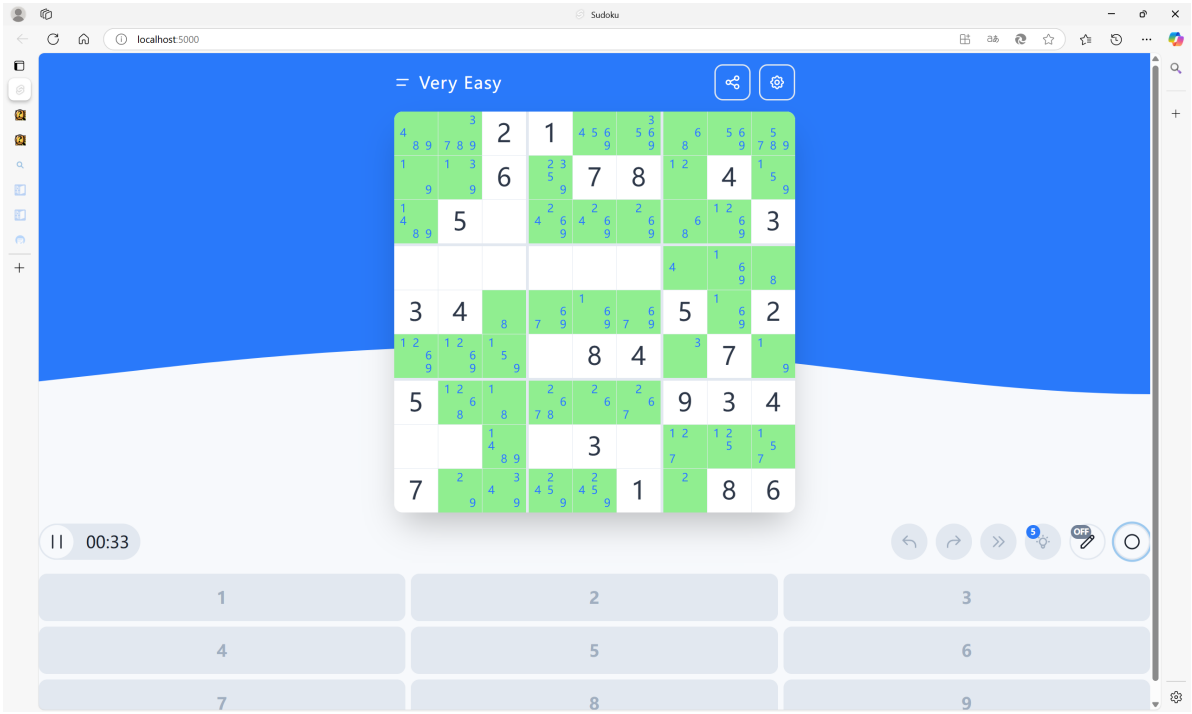


回溯到分支点，数字1变为灰色，不可以再次进入此分支

提示

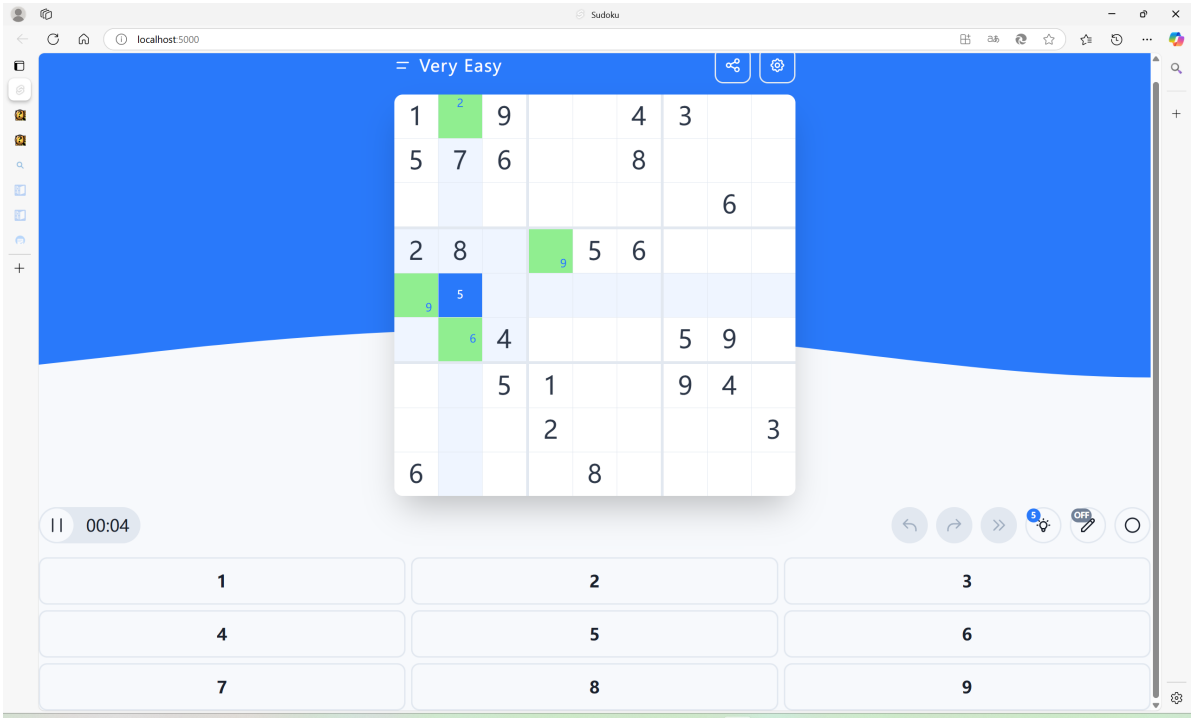


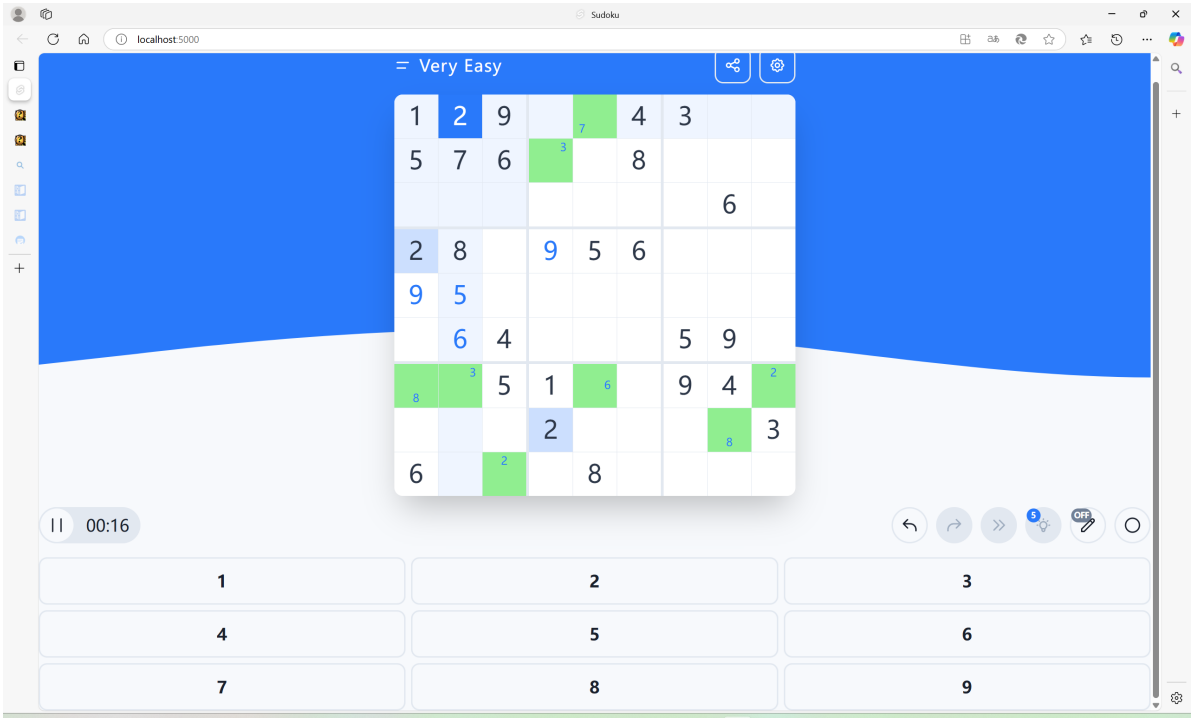
只显示候选值唯一的提示



显示多候选值的提示，通过最后一个按钮改变显示候选值的长度要求

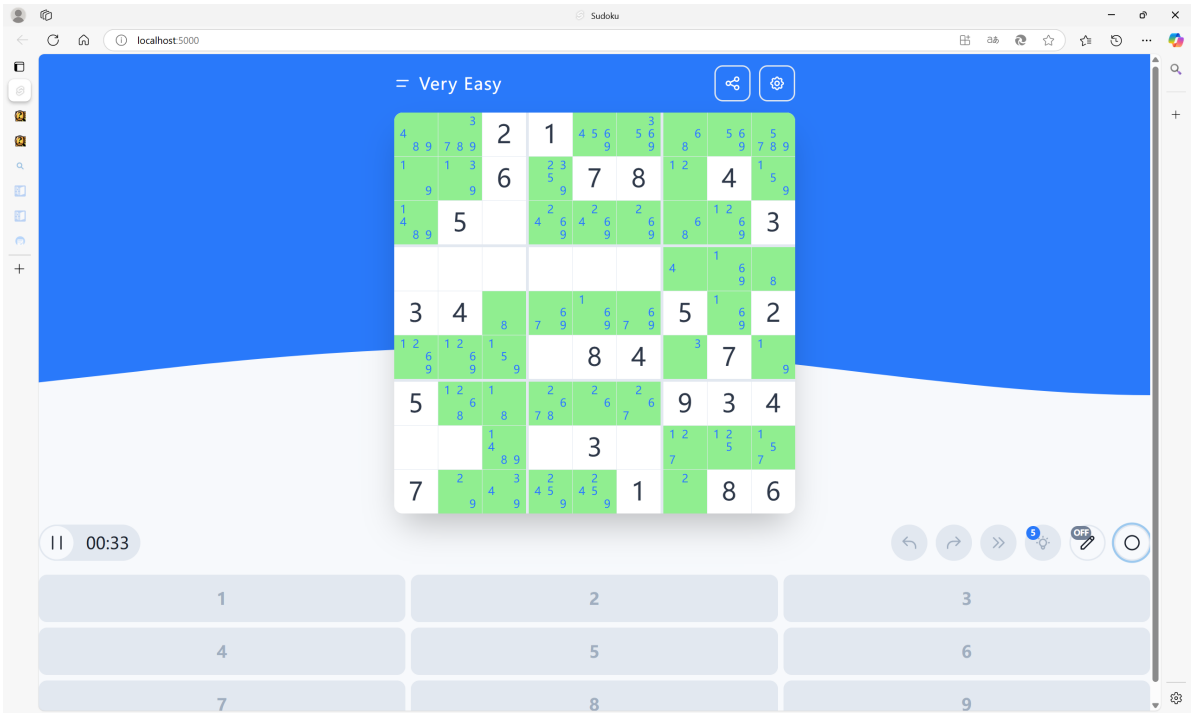
实时更新



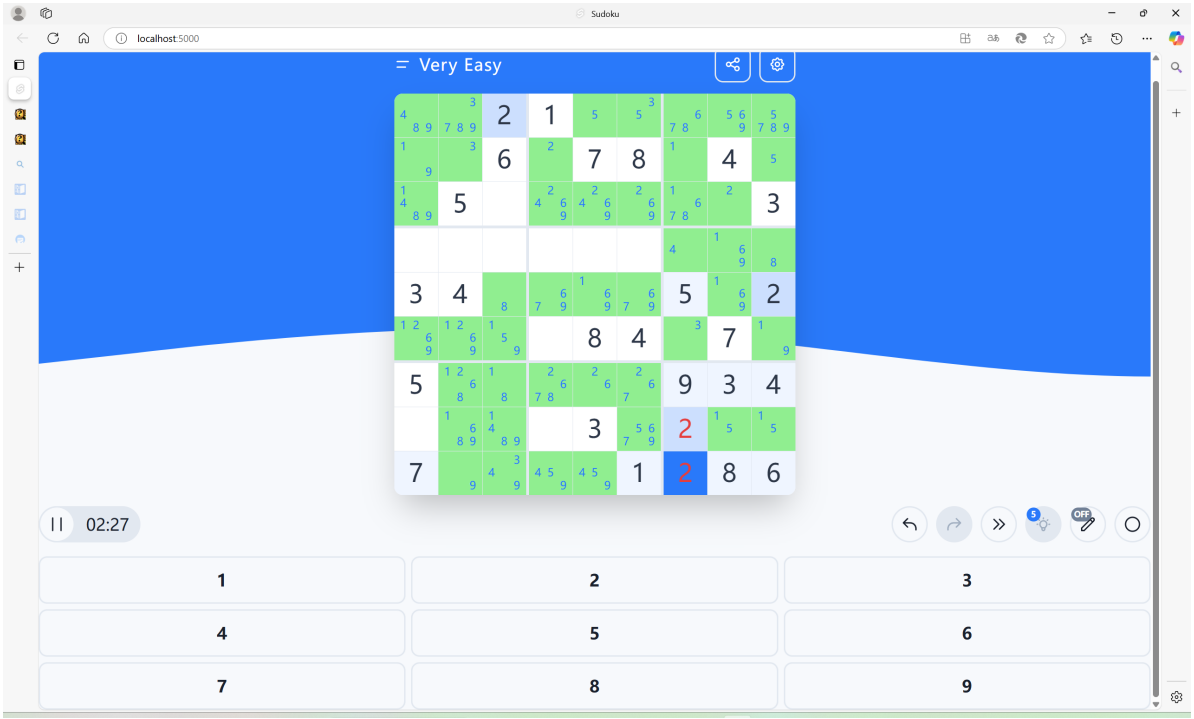


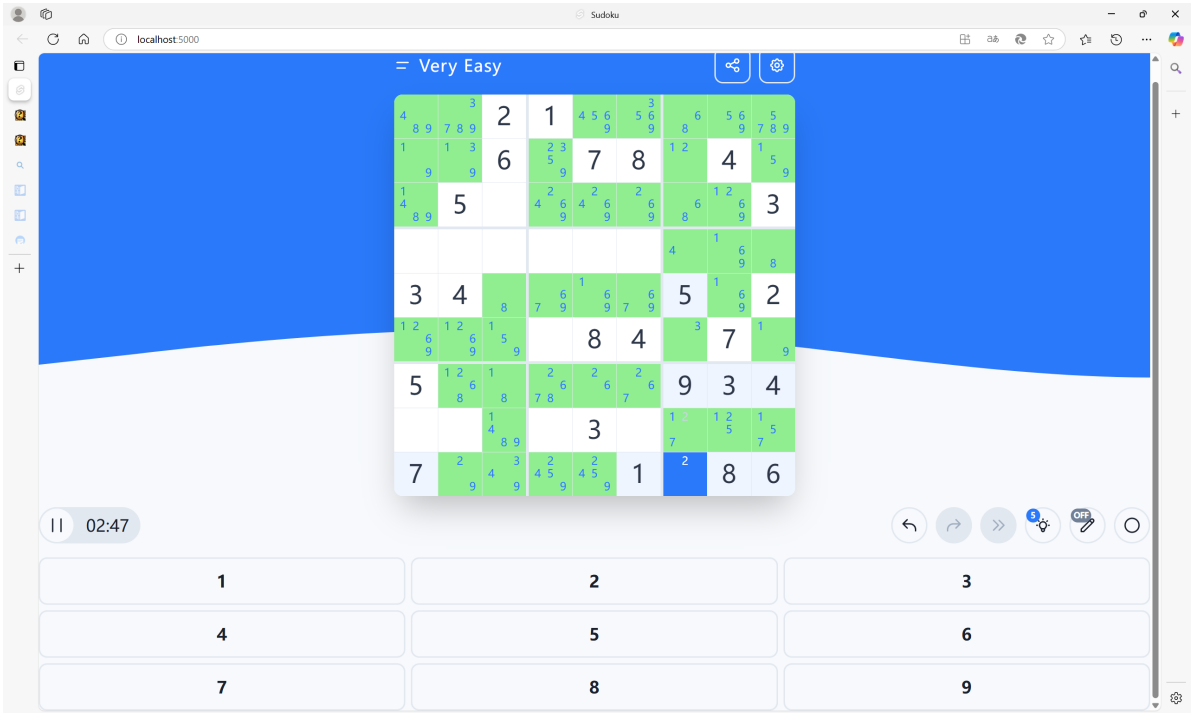
棋盘的变化会引起候选值的更新

回溯提示



可以通过点击候选值建立分支

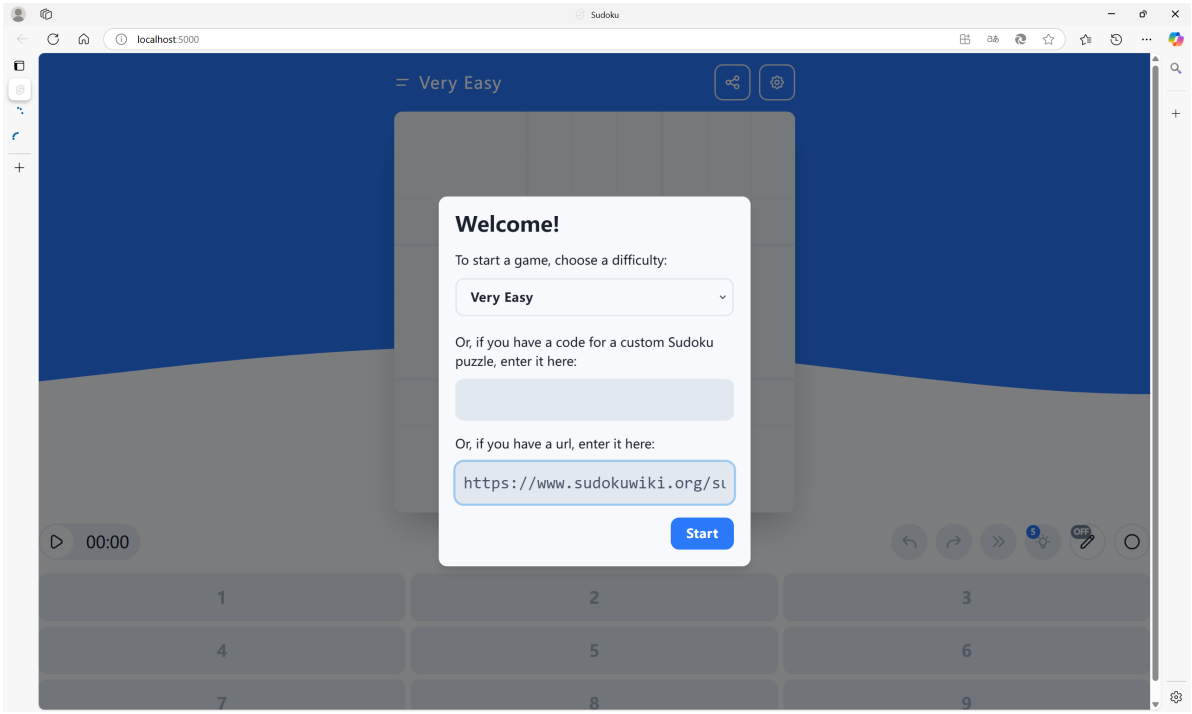


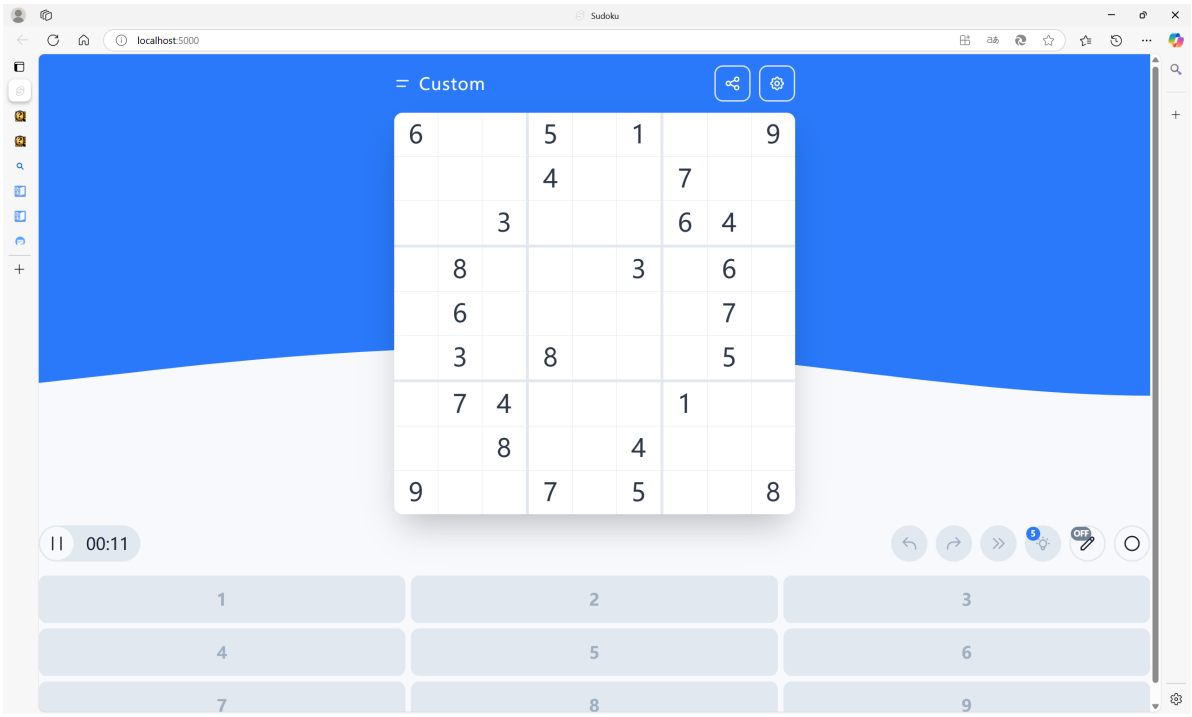


实现回溯

资源集成

题目导入


Sudoku
localhost:5000
Very Easy
Welcome!
To start a game, choose a difficulty:
Very Easy
Or, if you have a code for a custom Sudoku puzzle, enter it here:
Or, if you have a url, enter it here:
https://www.sudokuwiki.org/su
Start
00:00
1
2
3
4
5
6
7
8
9


Sudoku
localhost:5000
Custom
6 5 1 9
4 7
3 6 4
8 3 6
6 7
3 8 5
7 4 1
8 4
9 7 5 8
00:11
1
2
3
4
5
6
7
8
9