МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ   
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт математики и информационных систем

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ И ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Допущен к защите

Заведующий кафедрой ПМИ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.В. Разова

**Автоматическая генерация и решение Судоку**

Курсовой проект по дисциплине  
«Проектная и научно-исследовательская деятельность»

Выполнил студент группы ФИб-2301-51-00     / В.Р.Кочкин /

Руководитель К. п. н., зав. кафедрой ПМИ     / Е.В.Разова /

Работа защищена с оценкой     \_\_\_\_.\_\_\_\_.2021 г.

Члены комиссии:     /     /

    /     /

Киров 2021

**Содержание**

[**Введение** 3](#_Toc74039459)

[**1.** **Обзор алгоритмов генерации и решения Судоку** 4](#_Toc74039460)

[**1.1** **Терминология** 4](#_Toc74039461)

[**1.2** **Правила и происхождение игры Судоку** 4](#_Toc74039462)

[**1.3** **Обзор аналогов** 5](#_Toc74039463)

[**1.4** **Алгоритм генерации игрового поля** 9](#_Toc74039464)

[**1.4.1 Описание алгоритма** 9](#_Toc74039465)

[**1.4.2** **Тестирование времени работы алгоритма генерации** 16](#_Toc74039466)

[**1.5** **Алгоритмы решения Судоку** 17](#_Toc74039467)

[**1.5.1** **Алгоритм поиска с возвратом (ПВ)** 17](#_Toc74039468)

[**1.5.2** **Модифицированный поиск с возвратом (МПВ)** 22](#_Toc74039469)

[**1.5.3** **Тестирование времени работы алгоритмов решения** 28](#_Toc74039470)

[**1.6** **Выводы по разделу** 33](#_Toc74039471)

[**2.** **Проектирование и реализация игрового приложения** 34](#_Toc74039472)

[**2.1** **Анализ требований** 34](#_Toc74039473)

[**2.2** **Обоснование выбора языка программирования** 35](#_Toc74039474)

[**2.3** **Программная реализация** 36](#_Toc74039475)

[**2.4** **Описание интерфейса и функций приложения** 42](#_Toc74039476)

[**2.4** **Тестирование основных методов программы** 48](#_Toc74039477)

[**2.5** **Выводы по разделу** 49](#_Toc74039478)

[**Заключение** 50](#_Toc74039479)

[**Библиографический список** 52](#_Toc74039480)

[**Приложения** 54](#_Toc74039481)

[**Приложение А. Листинг программы** 54](#_Toc74039482)

[**Приложение Б. Листинг Unit-тестов** 60](#_Toc74039483)

# **Введение**

В мире с давних времен существует огромное количество головоломок, каждая из которых по-своему уникальна. Многие из них очень популярны, однако, одна из самых узнаваемых – японская головоломка Судоку. Она настолько популярна, что печаталась почти в каждой газете и даже выпускалась в отдельных сборниках Судоку. Люди решали эту головоломку на бумаге, чтобы скоротать время. Газеты уходят в прошлое, и теперь люди решают Судоку на смартфонах и компьютерах. Эта головоломка остается актуальна и по сей день, существует много реализаций игровых приложений, отличающихся не только дизайном, но и функционалом.

Таким образом, актуальность данной работы заключается во внесении нового функционала в компьютерную игру Судоку на основе анализа имеющихся аналогов.

Целью работы является выявление недостающего функционала аналогов и его реализация в среде программирования.

Для достижения указанной цели необходимо решить следующие задачи:

* провести анализ литературы;
* выполнить анализ существующих аналогов компьютерной реализации игры;
* сделать обзор алгоритмов генерации и решения Судоку;
* провести тесты времени работы алгоритмов;
* написать программную реализацию приложения;
* провести тесты работоспособности приложения.

# **Обзор алгоритмов генерации и решения Судоку**

## **Терминология**

Для упрощения работы с головоломкой вводятся следующие термины и обозначения:

* Размер Судоку – *N*;
* Поле – матрица Судоку размером *N×N*;
* Блок – часть матрицы размером *n×n*, где ;
* Полная ячейка – клетка матрицы, для которой, по условиям задачи или при решении алгоритмом, установлено значение;
* Пустая ячейка – клетка матрицы, для которой значение ещё не найдено;
* Список кандидатов – список возможных значений для пустой ячейки.

## **Правила и происхождение игры Судоку**

Судоку – головоломка, обычно состоящая из поля *N×N* клеток, разделенного на N квадратов (регионов, блоков), в котором каждая из *N* цифр встречается ровно один раз в каждой строке, столбце и в блоке. В зависимости от сложности скрывается *m* клеток, и целью головоломки является заполнение пустых клеток, соблюдая все вышеперечисленные правила. Правильно составленное Судоку должно иметь только одно решение. Наиболее частым вариантов является головоломка 9×9.

В настоящее время существует огромное количество модифицированных версий Судоку, например, Судоку с символами, “Перегородки”, “Куб”, “Цепочки” и т. д. [[9]](#_Библиографический_список) В данной работе рассматривается только классическое Судоку.

Обращаясь к истории головоломки, можно узнать, что, вопреки предрассудкам, Судоку вовсе не японский кроссворд. Предшественником и базисом игры был латинский квадрат, известный благодаря математику Л.Эйлеру, который использовал его для решения математических задач. Первая распространенная версия игры, похожая на современную, появилась в 1979г. В США под названием “Number Place”. Головоломка была модифицирована путем добавления в неё блоков 3×3, что сделало её более интересной.

Но современное название “Sudoku” дало японское издательство “Nicoli” в 1984 г. как сокращение фразы “Suuji wa dokushin ni kagiru”, что в переводе означает "число должно быть единственным". Так как в японском языке используются иероглифы, то в нём игры из серии кроссвордов невозможны, поэтому числовая игра Судоку стала наиболее популярна именно в Японии. Далее игра вернулась снова в США и разлетелась по всему миру. [[3](#_Библиографический_список)]

## **Обзор аналогов**

Для постановки задачи на обзор были выбраны три WEB-реализации игры Судоку. Проведя анализ функциональности, был выявлен список существующих и недостающих функций в каждой реализации.

* Первый аналог [[7](#_Библиографический_список)]

Данная реализация имеет: функцию отмены хода, выбор уровня сложности, проверку на ошибки (подсветка одинаковых цифр в строках, столбцах и блоках) и таймер.

К недостаткам можно отнести отсутствие: паузы, функции карандаша, подсказок, выбора размера поля, возможности решения собственного Судоку.

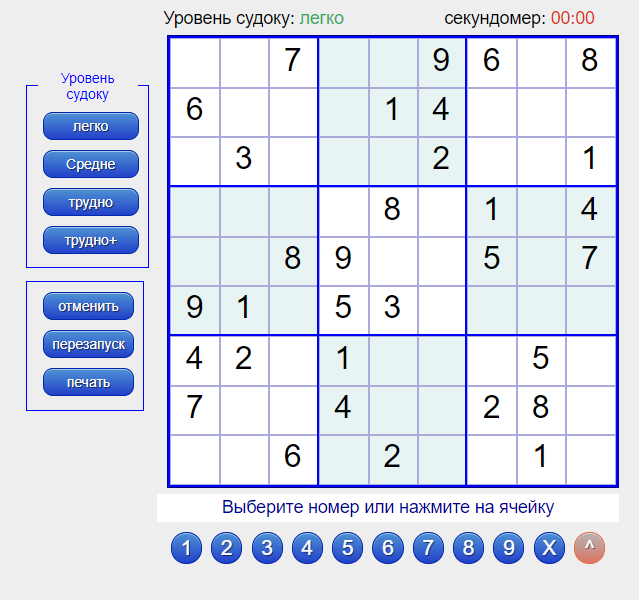


Рис. 1.3.1 – Первый аналог

* Второй аналог [[4](#_Библиографический_список)]

Автор Easybrain. Easybrain является разработчиком и издателем мобильных приложений, среди которых и Sudoku — наиболее популярная версия игры в App Store и Google Play. Также Easybrain является владельцем игрового сайта.

Данная реализация имеет: выбор уровня сложности, функцию карандаша и отмены хода, проверку на ошибки (подсветка одинаковых цифр в строках, столбцах и блоках), функцию подсказки, паузу и таймер.

К недостаткам можно отнести отсутствие: функции выбора размера поля, возможности решения собственного Судоку.

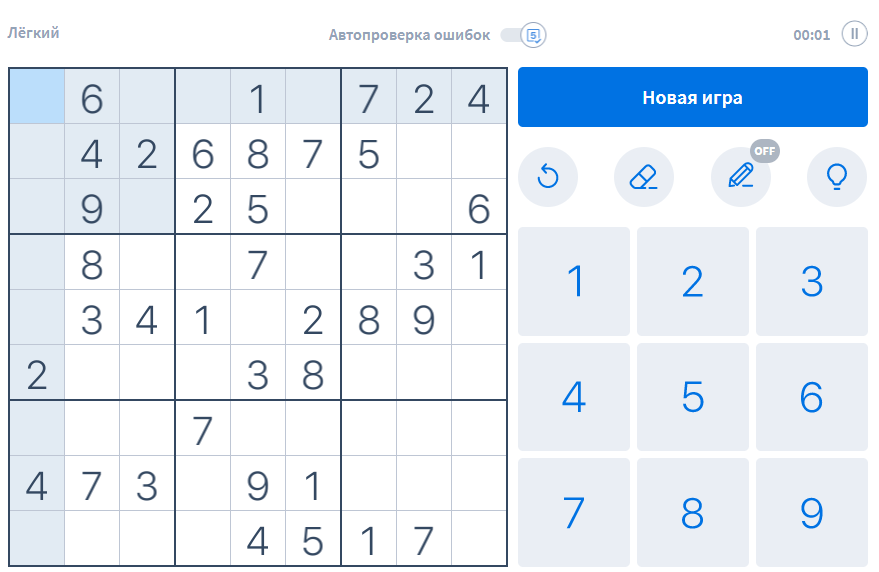


Рис. 1.3.2 – Второй аналог

* Третий аналог [[10](#_Библиографический_список)]

Автор Jtaugner. Игра выложена на платформе ЯндексИгры.

Данная реализация имеет: выбор размера поля (только 4×4 и 9×9), выбор уровня сложности, очистка ячейки, возможность показать всё решение, таймер.

К недостаткам можно отнести отсутствие: функции карандаша, подсказок, возможности решения собственного Судоку и отсутствие паузы.

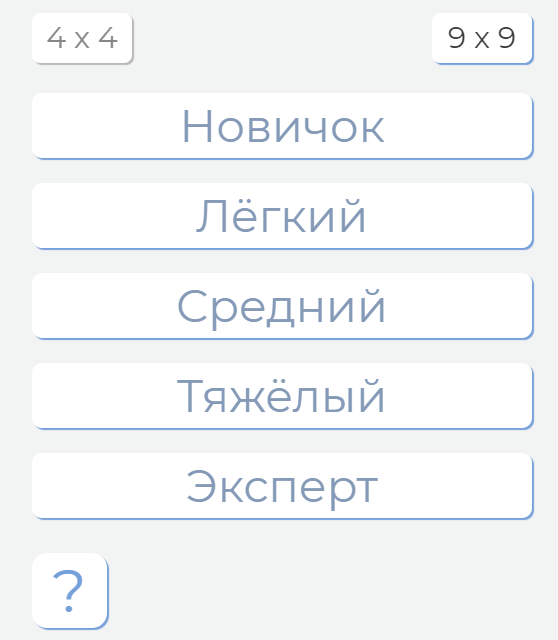


Рис. 1.3.3 – Меню третьего аналога

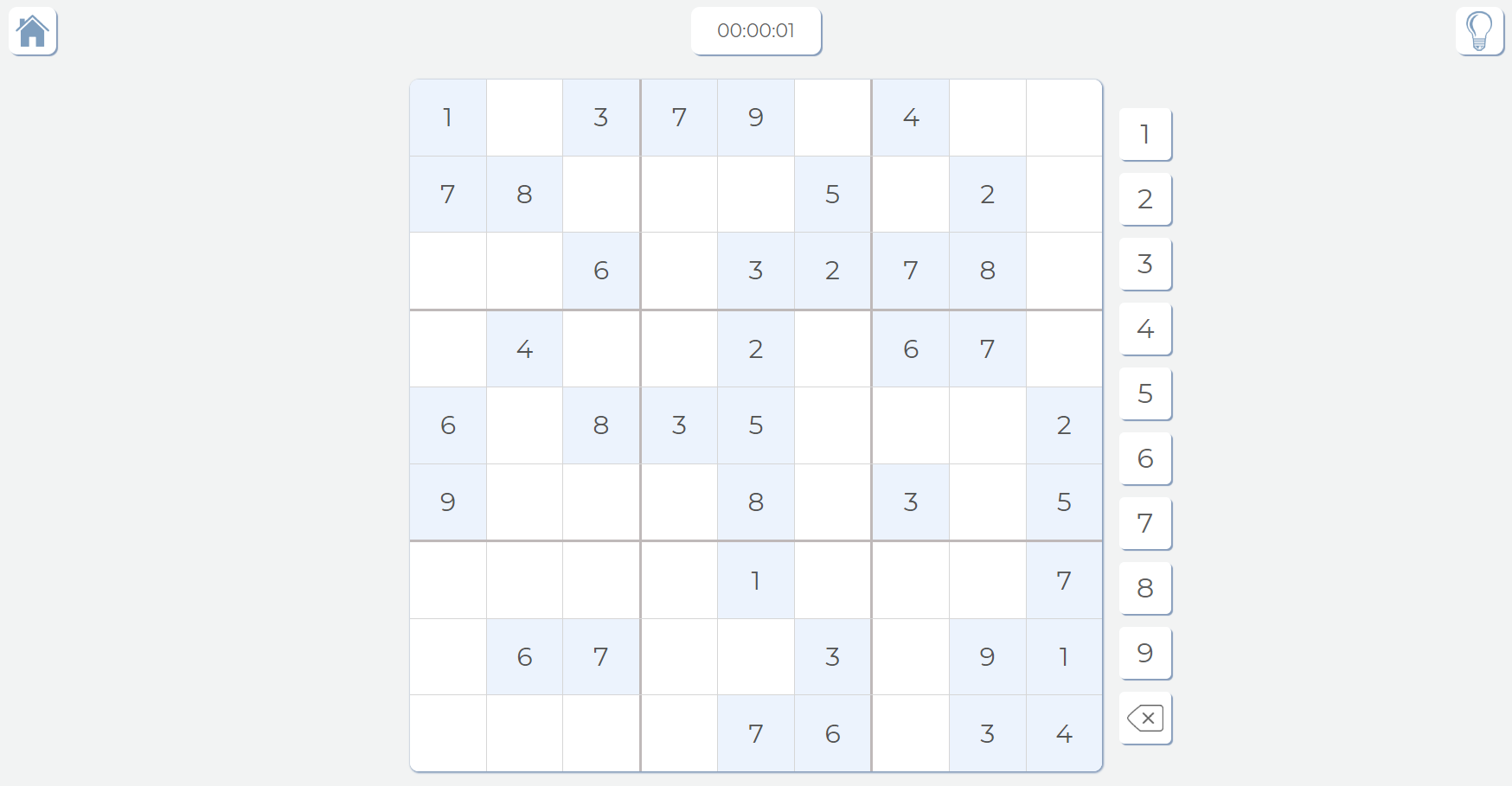


Рис. 1.3.4 – Третий аналог

Для визуального восприятия все плюсы и минусы приложений были внесены в таблицу 1.1.

Таблица 1.1 – Сравнение функционала аналогов

|  |  |  |  |
| --- | --- | --- | --- |
| Функция | Номер аналога | | |
| 1 | 2 | 3 |
| Выбор размера поля | - | - | + |
| Выбор сложности | + | + | + |
| Функция подсказки | - | + | - |
| **Возможность решения собственного Судоку** | **-** | **-** | **-** |
| Функция карандаша | - | + | - |
| Показать решение | - | - | + |
| Отмена хода | + | - | + |
| Пауза | - | + | - |
| Таймер | + | + | + |

Проанализировав функционал каждой реализации, можно сделать вывод, что такой функции, как возможность решения собственного Судоку, не реализовано ни в одном из аналогов. Данная функция нужна для людей, которые хотят ввести собственное Судоку, например, из газеты, чтобы попытаться решить его на компьютере или узнать правильное решение.

Таким образом, задачей данной работы является анализ и выбор алгоритмов генерации и решения Судоку, а также реализация игровой программы, содержащую лучшие стороны аналогов и функцию решения собственного Судоку.

## **Алгоритм генерации игрового поля**

### **1.4.1 Описание алгоритма**

Для генерации поля за основу была взята смещенная базовая сетка, подчиняющаяся игровым правилам, которая выглядит следующим образом:

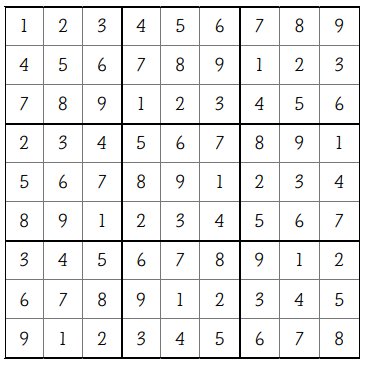


Рис. 1.4.1 – Базовая сетка

Данное поле строится по формуле:

|  |  |
| --- | --- |
|  | (1.1) |

Далее его следует перемешать, но, чтобы корректность поля не была нарушена. Для этого понадобятся следующие функции, представленные в виде псевдокода:

* *SwapBlockInRow* – меняет местами две случайных строки из блоков (см. рис. 1.4.1). Пример работы приведен на рис. 1.4.2.

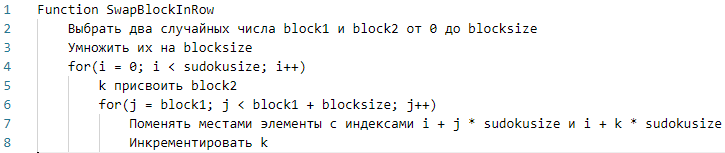


Рис. 1.4.1 – Функция *SwapBlockInRow*

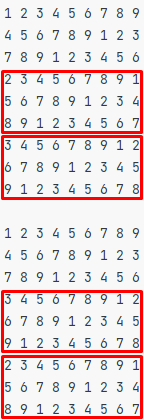


Рис. 1.4.2 – Пример работы функции *SwapBlockInRow*

* *SwapBlockInColumn* – меняет местами два случайных столбца из блоков (см. рис. 1.4.3). Пример работы приведен на рис. 1.4.4.

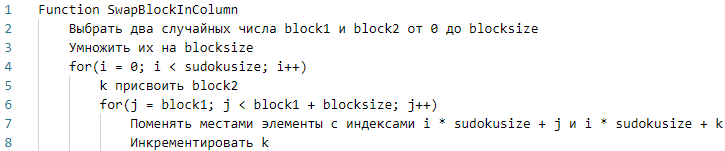


Рис. 1.4.3 – Функция *SwapBlockInColumn*

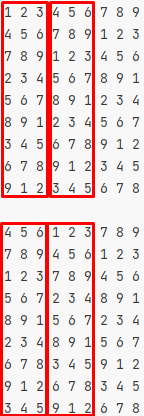


Рис. 1.4.4 – Пример работы функции *SwapBlockInColumn*

* *SwapRowsInBlock* – меняет местами две случайные строки в пределах блока (см. рис. 1.4.5). Пример работы приведен на рис. 1.4.6.

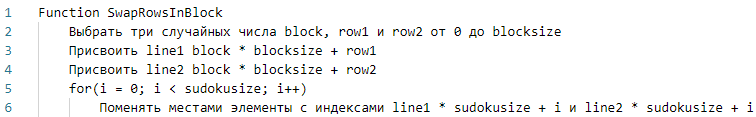


Рис. 1.4.5 – Функция *SwapRowsInBlock*

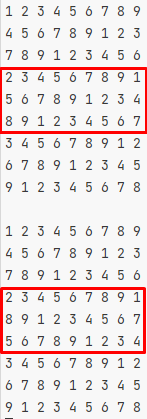


Рис. 1.4.6 – Пример работы функции *SwapRowsInBlock*

* *SwapColumnsInBlock* – меняет местами два случайных столбца в пределах блока (см. рис. 1.4.7). Пример работы приведен на рис. 1.4.8.

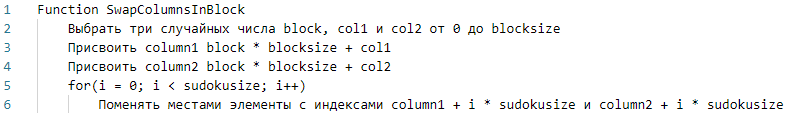


Рис. 1.4.7 – Функция *SwapColumnsInBlock*

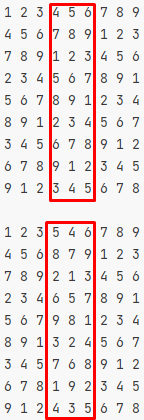


Рис. 1.4.8 – Пример работы функции *SwapColumnsInBlock*

* *MatrixTransposition* – транспонирует матрицу, то есть превращает строки в столбцы и наоборот (см. рис. 1.4.9). Пример работы приведен на рис. 1.4.10.

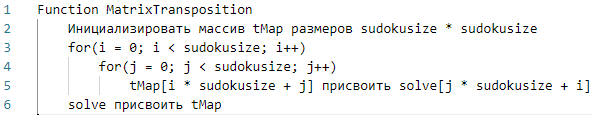


Рис. 1.4.9 – Функция *MatrixTransposition*

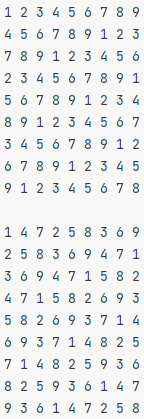


Рис. 1.4.10 – Пример работы функции *MatrixTransposition*

Благодаря данным преобразованиям, сетка Судоку будет перемешена и не потеряет свою корректность. [[1](#_Библиографический_список)]

Чтобы перемешать поле, следует случайно вызывать данные функции много раз. Для выбора оптимального количества случайных перемешиваний были проведены наблюдения за сгенерированным полем Судоку при разных количествах итераций перемешивания.

Для начала было взято 100 итераций, после которых игровое поле различных размеров выглядело следующим образом (см. рис, 1.4.11 – 1.4.14):



Рис. 1.4.11 – Судоку 4×4 100 итераций

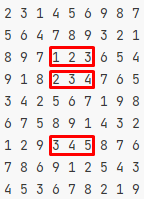


Рис. 1.4.12 – Судоку 9×9 100 итераций

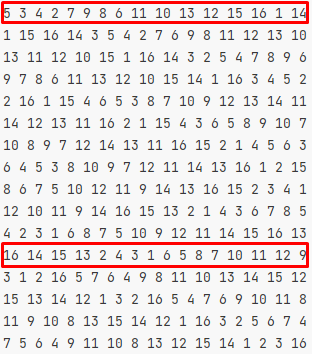


Рис. 1.4.13 – Судоку 16×16 100 итераций

Из скриншотов видно, что поле размером 4×4 вполне неплохо перемешалось, а вот поля большего размера содержат в себе цепочки по 3 и более подряд идущих цифр, поэтому для них требуется большее количество итераций. Таким образом, для генерации поля 4×4 было выбрано 100 итераций перемешивания.

Далее подобным образом было выбрано оптимальное количество итераций для каждого размера Судоку, а именно: 9×9 – 1000 итераций, 16×16 – 5000 итераций, 25×25 – 10000 итераций.

Исходя из того, что функции перемешивания матрицы работают за *O(N2)*, а также при генерации происходит *k* итераций перемешиваний, где *k* – это константа, то общую временную сложность алгоритма можно представить как *O(N2)*.

### **Тестирование времени работы алгоритма генерации**

Базой для тестирования стали Судоку размера 4×4, 9×9, 16×16, 25×25. Для каждого размера было произведено 50 запусков и высчитан средний результат времени генерации поля. Получившиеся данные были занесены в таблицу, по которой построен график зависимости времени работы алгоритма от размера головоломки (см. рис. 1.4.14).

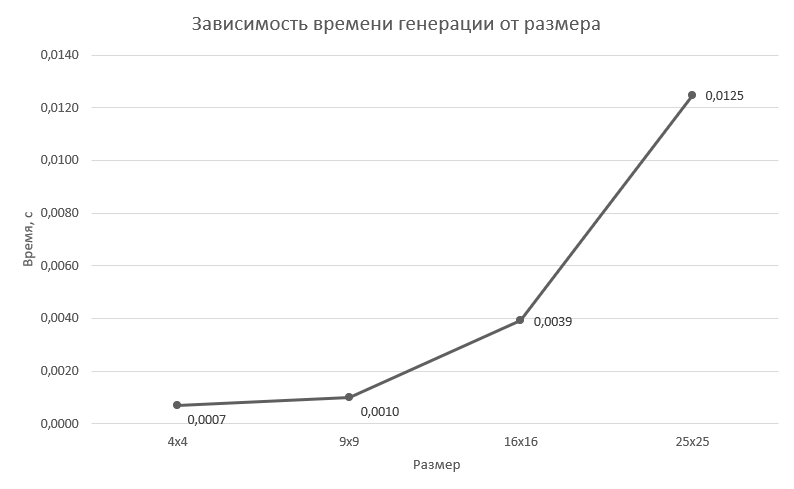


Рис. 1.4.14 – График времени генерации Судоку

Тесты проводились в IDE JetBrains Rider 2021.1.2, Build #RD-211.7142.19, built on April 22, 2021 на компьютере со следующими основными характеристиками:

* Процессор: Ryzen 5 3600 6/12 4.2 Гц;
* Оперативная память: 16 ГБ 3600 МГц;
* Накопитель: SSD 3500 МБ/сек. чтение и 3300 МБ/сек. запись;

Для замера времени работы алгоритмов использовалась библиотека *System.Diagnostics*, а именно класс *Stopwatch*.

Таким образом, предложенный алгоритм довольно неплохо справляется со своей задачей генерации игрового поля разных размеров за приемлемое время.

## **Алгоритмы решения Судоку**

### **Алгоритм поиска с возвратом (ПВ)**

Самым прямолинейным алгоритмом решения Судоку является полный перебор всех вариантов для каждой его ячейки. Идея алгоритма заключается в следующих четырёх этапах:

1. выбор пустой ячейки и присваивание ей кандидата;
2. проверка текущего поля на корректность заполнения;
3. если поле корректно, то продолжение алгоритма для следующей пустой ячейки;
4. если некорректно – обнуление ячейки и переход к следующему кандидату.

Если алгоритм переберет для ячейки всех кандидатов и поле будет некорректно, то данное Судоку не имеет решения. Если же алгоритм заполнит все пустые ячейки, то решение будет найдено.

Алгоритм можно представить в виде псевдокода (см. рис. 1.5.1), где *index* – номер рассматриваемой ячейки, *A* – одномерный массив значений Судоку, в котором пустые ячейки обозначены нулем;   
*ChekPossibleValue(val, index)* – функция проверки значения *val* в ячейке *index* на соответствие правилам Судоку (см. рис. 1.5.2). [[6](#_Библиографический_список)]



Рис. 1.5.1 – Поиск с возвратом

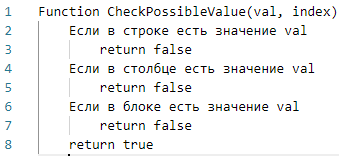


Рис. 1.5.2 – Функция *CheckPossibleValue*

Исходя из того, что алгоритм для каждой пустой ячейки *m* перебирает всех кандидатов *N*, то временную сложность алгоритма можно оценить как *O(Nm)*.

Для наглядности работы алгоритма приведем пример. Пусть на вход алгоритма поступает следующее Судоку (см. рис. 1.5.3).

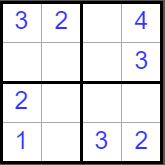


Рис. 1.5.3 – Судоку 4×4

Так как алгоритм запускается от 0-й ячейки, то он будет рекурсивно запускаться от index + 1 ячейки, пока не дойдёт до первой пустой. Все шаги пропуска непустых ячеек будут опущены.

1. Допустим, алгоритм находится на ячейке с индексом 2. Ячейке присваивается число 1 (см. рис. 1.5.4). Судоку проходит проверку на валидность, и алгоритм рекурсивно вызывает себя от следующей ячейки.

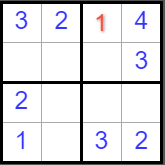


Рис. 1.5.4 – Заполнение ячейки с индексом 2

1. Алгоритм в ячейке 4. Ячейке присваивается число 1. Судоку не проходит проверку на валидность. Ячейке присваивается число 2. Опять не проходит. Число 3 – не проходит. Наконец, число 4 – Судоку проходит проверку на валидность (см. рис. 1.5.5), и алгоритм запускается для следующей ячейки.

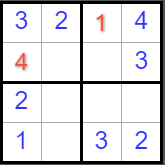


Рис. 1.5.5 – Заполнение ячейки с индексом 4

1. Ячейка 5. Ей присваивается значение 1. Судоку проходит проверку (см. рис. 1.5.6), алгоритм запускается для следующей ячейки.

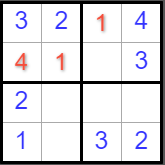


Рис. 1.5.6 – Заполнение ячейки с индексом 5

1. Ячейка 6. Присваивается число 1. Судоку не проходит проверку. Число 2 – проверку проходит (см. рис. 1.5.7), запускается для следующей ячейки.

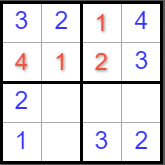


Рис. 1.5.7 – Заполнение ячейки с индексом 6

1. Ячейка 9. Число 1 – не проходит, 2 – не проходит, 3 – проходит (см. рис. 1.5.8). Запускается для следующей ячейки.

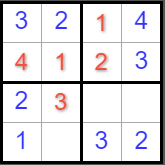


Рис. 1.5.8 – Заполнение ячейки с индексом 9

1. Ячейка 10. 1 не подходит, 2 не подходит, 3 не подходит, 4 подходит (см. рис. 1.5.9). Запускается для следующей ячейки.

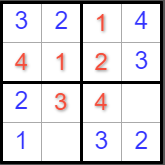


Рис. 1.5.9 – Заполнение ячейки с индексом 10

1. Ячейка 11. 1 проходит проверку (см. рис. 1.5.10). Запускается для следующей ячейки.

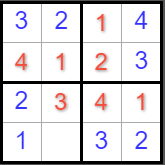


Рис. 1.5.10 – Заполнение ячейки с индексом 11

1. Ячейка 13. 1 не подходит, 2 не подходит, 3 не подходит, 4 подходит (см. рис. 1.5.11). Так как были заполнены все пустые ячейки и корректность Судоку не была нарушена, то алгоритм завершает свою работу.

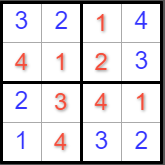


Рис. 1.5.11 – Заполнение ячейки с индексом 13

### **Модифицированный поиск с возвратом (МПВ)**

В отличие от обычного поиска с возвратом его модификация перебирает не все возможные варианты для ячейки, а только тех кандидатов, которые не противоречат условиям. Также вместо ближайшей пустой ячейки выбирается не следующая по порядку, а следующая ячейка с наименьшим числом кандидатов.

Алгоритм представлен в виде псевдокода (см. рис. 1.5.12), где *CountCandidats* – функция, которая считает количество кандидатов для каждой ячейки матрицы (см. рис. 1.5.13), а *ModifiedBruteForceWithReturnRecursion()* – рекурсивная функция подбора решения Судоку, аргументом которой является функция *MinСandidates*, возвращающая индекс ячейки с минимальным количеством кандидатов (см. рис. 1.5.14 и рис. 1.5.15 соотв.). [[6](#_Библиографический_список)]

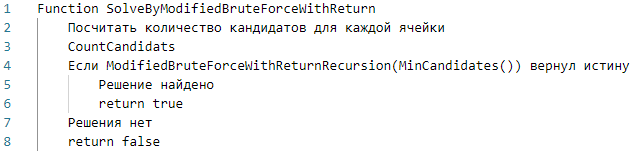


Рис. 1.5.12 – Модифицированный поиск с возвратом

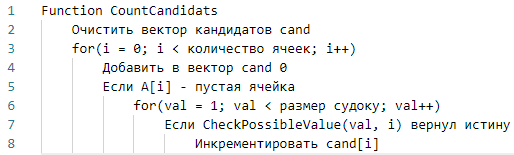


Рис. 1.5.13 – Функция *CountCandidats*

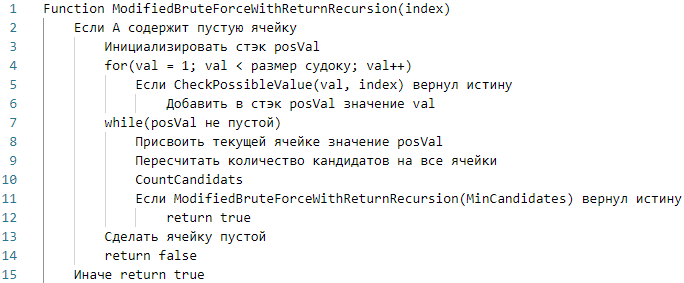


Рис. 1.5.14 – Функция *ModifiedBruteForceWithReturnRecursion*

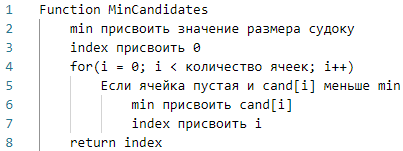


Рис. 1.5.15 – Функция *MinСandidates*

Для наглядности работы алгоритма приведем пример. Пусть на вход алгоритма поступает следующее Судоку (см. рис. 1.5.16).

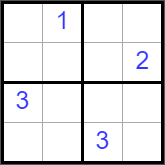


Рис. 1.5.16 – Судоку 4×4

1. Заполняется массив количества кандидатов *cand* = {2, 0, **1**, 2, 1, 2, 2, 0, 0, 2, 3, 2, 3, 2, 0, 2}.
2. Алгоритм запускается от ячейки с минимальным количеством кандидатов, не равным нулю, то есть от 2-й ячейки. Заполняется стэк возможных кандидатов для этой ячейки, который содержит только цифру 4. Ячейке присваивается значение 4 (см. рис. 1.5.17), пересчитывается массив *cand* = {**1**, 0, 0, 1, 1, 2, 1, 0, 0, 2, 2, 2, 3, 2, 0, 2}. Алгоритм запускается от ячейки с индексом 0.

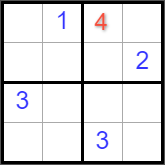


Рис. 1.5.17 – Заполнение ячейки с индексом 2

1. Ячейка 0. Ей присваивается значение 2 (см. рис. 1.5.18). Массив *cand* = {0, 0, 0, **1**, 1, 2, 1, 0, 0, 2, 2, 2, 2, 2, 0, 2}. Алгоритм запускается для ячейки с индексом 3.

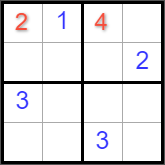


Рис. 1.5.18 – Заполнение ячейки с индексом 0

1. Ячейка 3. Ей присваивается значение 3 (см. рис. 1.5.19). Массив *cand* = {0, 0, 0, 0, **1**, 2, 1, 0, 0, 2, 2, 2, 2, 2, 0, 2}. Алгоритм запускается для ячейки с индексом 4.

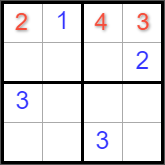


Рис. 1.5.19 – Заполнение ячейки с индексом 3

1. Ячейка 4. Ей присваивается значение 4 (см. рис. 1.5.20). Массив *cand* = {0, 0, 0, 0, 0, **1**, 1, 0, 0, 2, 2, 2, 1, 2, 0, 2}. Алгоритм запускается для ячейки с индексом 5.



Рис. 1.5.20 – Заполнение ячейки с индексом 4

1. Ячейка 5. Ей присваивается значение 3 (см. рис. 1.5.21). Массив *cand* = {0, 0, 0, 0, 0, 0, **1**, 0, 0, 2, 2, 2, 1, 2, 0, 2}. Алгоритм запускается для ячейки с индексом 6.

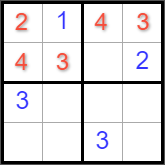


Рис. 1.5.21 – Заполнение ячейки с индексом 5

1. Ячейка 6. Ей присваивается значение 1 (см. рис. 1.5.22). Массив *cand* = {0, 0, 0, 0, 0, 0, 0, 0, 0, 2, **1**, 2, 1, 2, 0, 2}. Алгоритм запускается для ячейки с индексом 10.

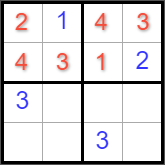


Рис. 1.5.22 – Заполнение ячейки с индексом 6

1. Ячейка 10. Ей присваивается значение 2 (см. рис. 1.5.23). Массив *cand* = {0, 0, 0, 0, 0, 0, 0, 0, 0, **1**, 0, 2, 1, 2, 0, 2}. Алгоритм запускается для ячейки с индексом 9.

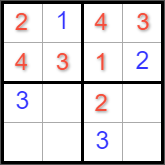


Рис. 1.5.23 – Заполнение ячейки с индексом 10

1. Ячейка 9. Ей присваивается значение 4 (см. рис. 1.5.24). Массив *cand* = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **1**, 1, 1, 0, 2}. Алгоритм запускается для ячейки с индексом 11.

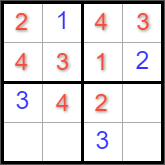


Рис. 1.5.24 – Заполнение ячейки с индексом 9

1. Ячейка 11. Ей присваивается значение 1 (см. рис. 1.5.25). Массив *cand* = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **1**, 1, 0, 1}. Алгоритм запускается для ячейки с индексом 12.



Рис. 1.5.25 – Заполнение ячейки с индексом 11

1. Ячейка 12. Ей присваивается значение 1 (см. рис. 1.5.26). Массив *cand* = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **1**, 0, 1}. Алгоритм запускается для ячейки с индексом 13.

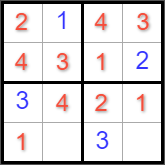


Рис. 1.5.26 – Заполнение ячейки с индексом 12

1. Ячейка 13. Ей присваивается значение 2 (см. рис. 1.5.27). Массив *cand* = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **1**}. Алгоритм запускается для ячейки с индексом 15.

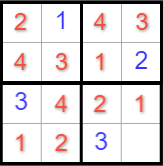


Рис. 1.5.27 – Заполнение ячейки с индексом 13

1. Ячейка 15. Ей присваивается значение 4 (см. рис. 1.5.28). Так как были заполнены все ячейки, алгоритм завершает свою работу.

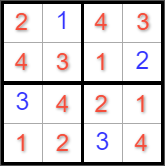


Рис. 1.5.28 – Заполнение ячейки с индексом 15

### **Тестирование времени работы алгоритмов решения**

Приведенные выше алгоритмы будут сравниваться по одному критерию – время работы. В качестве исходных данных были взяты 5 разных Судоку каждого размера и сложности, а именно: 4×4, легкое 9×9, среднее 9×9, сложное 9×9, легкое 16×16, среднее 16×16, легкое 25×25. Готовые Судоку были взяты с электронного ресурса PlaySudoku [[2](#_Библиографический_список)].

В ходе тестирования результаты записывались в таблицу в Microsoft Excel, в которой по получившимся данным были построены диаграммы, представленные ниже (см. рис. 1.5.28–1.5.32). Для каждого Судоку и алгоритма было проведено 10 тестовых запусков, чтобы получить среднее значение. На оси абсцисс представлены 5 разных Судоку данного размера, на оси ординат – время в секундах. Черным цветом обозначен обычный поиск с возвратом, а серым – модифицированный.

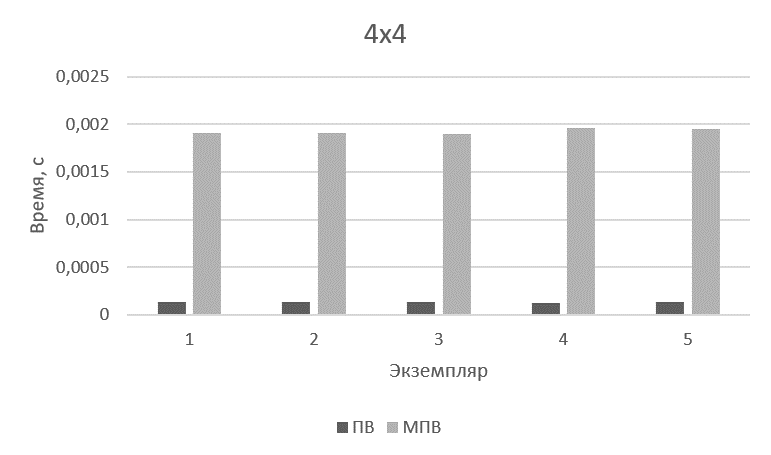


Рис. 1.5.28 – Диаграмма времени решения Судоку 4×4

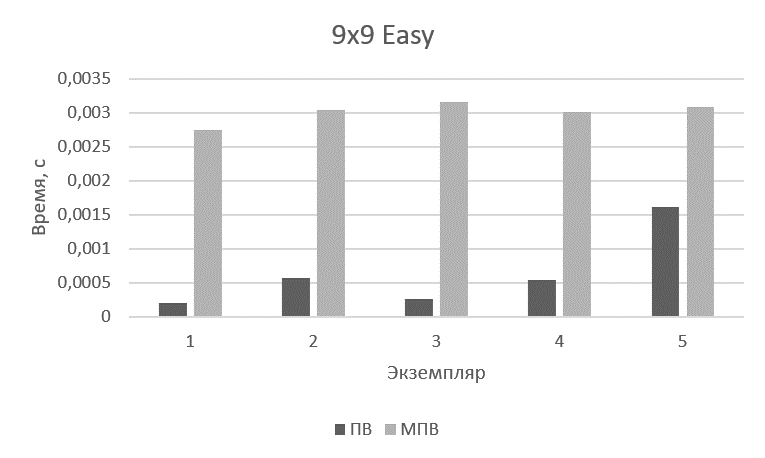


Рис. 1.5.29 – Диаграмма времени решения легкого Судоку 9×9

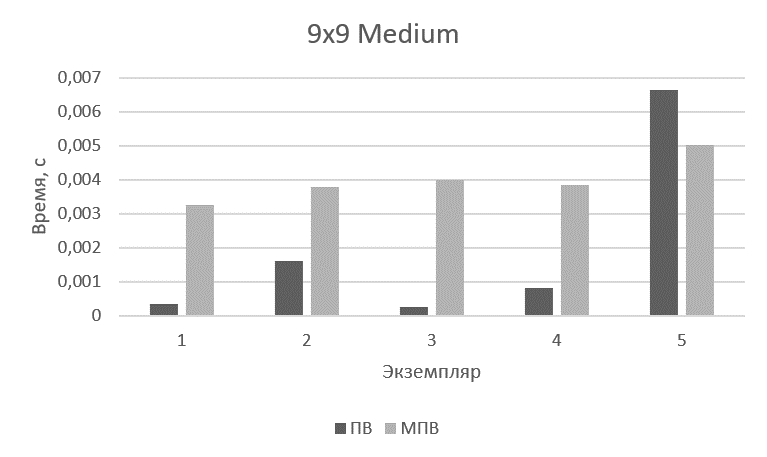


Рис. 1.5.30 – Диаграмма времени решения среднего Судоку 9×9

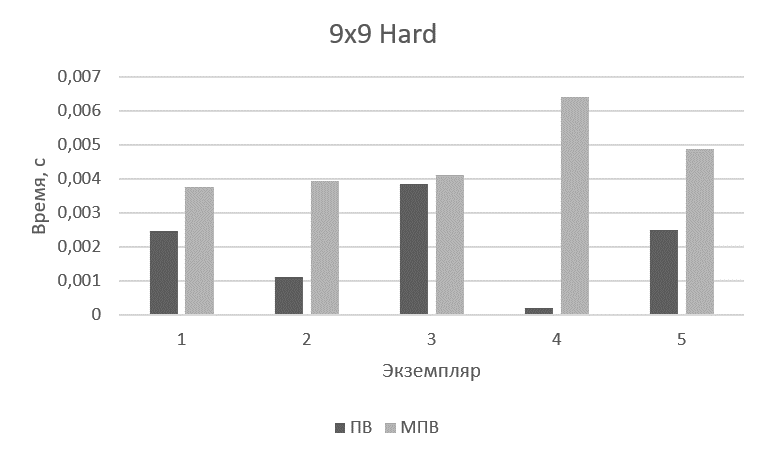


Рис. 1.5.31 – Диаграмма времени решения сложного Судоку 9×9

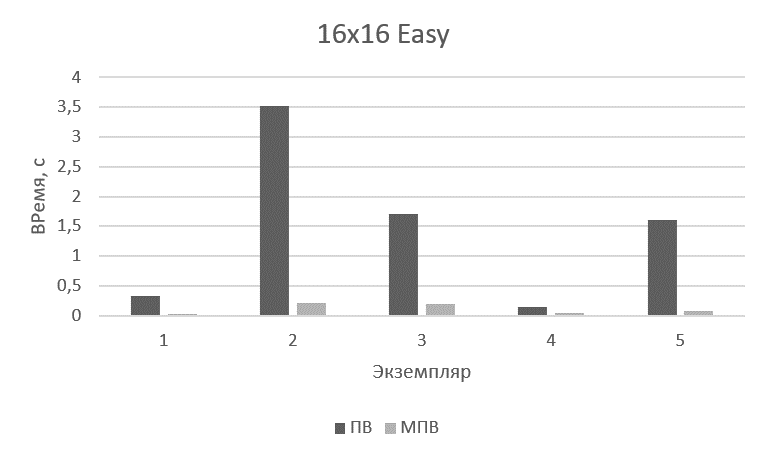


Рис. 1.5.32 – Диаграмма времени решения легкого Судоку 16×16

Чтобы было более наглядно, для Судоку каждого размера и сложности было вычислено среднее значение времени работы алгоритмов и построен общий график (см. рис. 1.5.33).

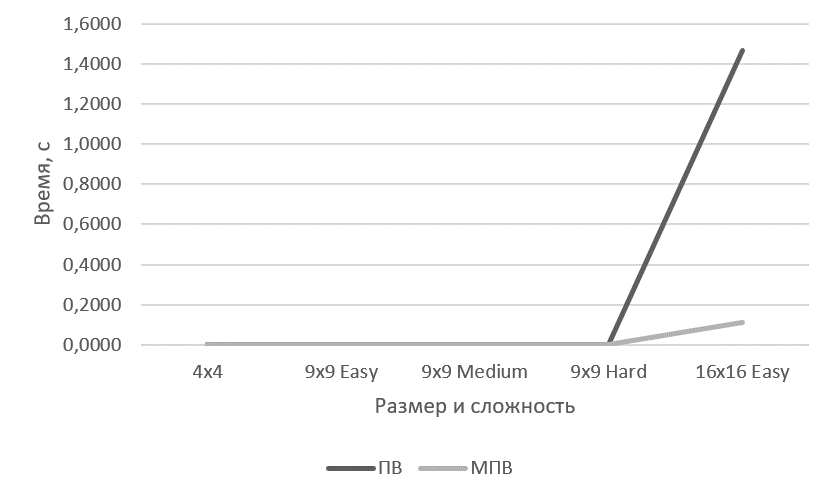


Рис. 1.5.33 – График сравнения двух алгоритмов ПВ и МПВ

Так как на промежутке от 4×4 до 9×9 Hard различия между алгоритмами не видно, то было решено построить для данного промежутка отдельный график (см. рис. 1.5.34).

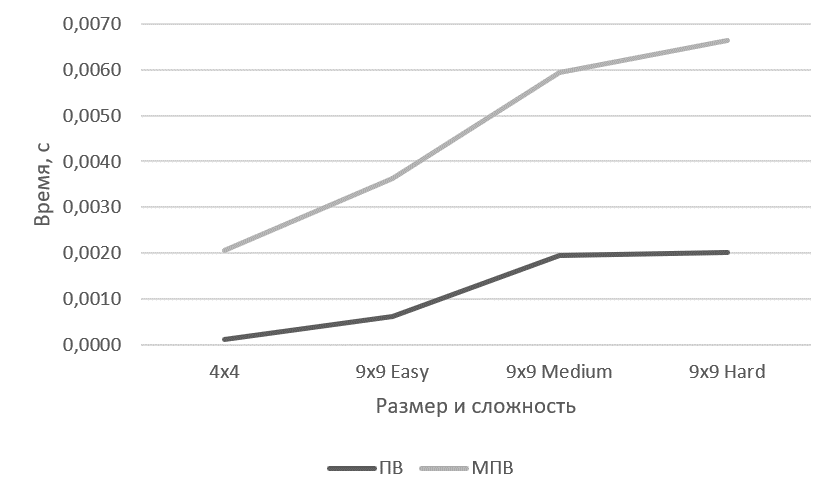


Рис. 1.5.34 – График сравнения алгоритмов на отрезке от 4×4 до 9×9

По результатам тестирования стало ясно, что Судоку размера 4×4 и 9×9 первый алгоритм решает быстрее, нежели второй. Второй же алгоритм показывал себя лучше на Судоку размером 16×16 легкой сложности, однако не все предложенные Судоку были решены алгоритмами за приемлемое время. Например, Судоку 16×16 средней и высокой сложности, а также Судоку 25×25 любой сложности вообще не были решены (время работы алгоритмов превышало 30 сек.). Это говорит о том, что данные алгоритмы не подходят для матриц размера 16×16 и более, потому что для каждой ячейки существует много кандидатов, и рекурсия не справляется с каждым из них.

Так как задачи реализовать приложение для решения Судоку больших размеров не стояло, то было решено в программной реализации для решения головоломки размеров 4×4 и 9×9 использовать алгоритм поиска с возвратом, а от решения Судоку больших размеров отказаться.

## **Выводы по разделу**

В данном разделе были рассмотрены аналоги игры Судоку и проведен обзор их функциональности. Данная информация была структурирована, благодаря чему был выявлен функционал, не реализованный ни в одном приложении. Исходя их этого была поставлена задача разработки игрового приложения, сочетающее в себе лучшие стороны каждого из аналогов, а также новую функцию – возможность решения собственного Судоку.

Также в разделе рассмотрен алгоритм генерации игрового поля Судоку и два алгоритма решения Судоку. Были проведены тесты, которые зафиксированы в таблице и на графиках, и сделан вывод о скорости работы алгоритмов.

# **Проектирование и реализация игрового приложения**

## **Анализ требований**

В главном меню игрового приложения у пользователя должна быть возможность выбрать размер (4×4, 9×9 для своей игры и 4×4, 9×9, 16×16, 25×25 для генерируемой) и уровень сложности игры (легкая, средняя, сложная, эксперт), посмотреть информацию об авторе, ознакомиться с правилами, выйти из программы, а также начать генерируемую игру или ввести поле самому.

Если пользователь начинает генерируемую игру, то открывается игровая форма с указанными параметрами размера и сложности головоломки. В данном окне пользователь должен увидеть выбранную сложность и таймер, а также иметь возможность регулировать громкость фоновой музыки и возможность начать игру. При старте игры в окне генерируется и выводится поле Судоку, в клетки которого игрок с помощью цифр на клавиатуре сможет вводить значения. Во время игры у пользователя должна быть возможность поставить игру на паузу, воспользоваться подсказкой, проверить решение на корректность и увидеть полное решение Судоку.

Чтобы воспользоваться подсказкой, пользователь должен выделить интересующую его клетку и нажать кнопку подсказки, после чего появится правильная цифра и ячейка заблокируется.

При нажатии на кнопку проверки решения, если решение правильное, должно появиться сообщение об этом и предложение о начале новой игры, если же решение неправильное, то приложение должно подсветить некорректные ячейки, чтобы игроку было понятно, где он ошибся.

При использовании функции показа решения Судоку, все пустые клетки поля должны заполниться верными цифрами, после чего пользователь сможет проверить корректность решения.

Если пользователь начинает собственную игру, то открывается вторая форма. Исходя из выбранного размера, на форме должно быть пустое поле игры, в которое пользователь должен ввести своё Судоку посредством клавиатуры. При нажатии на кнопку старта игры, программа должна проверить введенное игроком Судоку на корректность и на решаемость. Если проверки пройдены, то введенные ячейки блокируются и игра начинается, иначе программа должна предупредить об ошибке в поле.

В данной форме должен быть доступен весь функционал с формы генерируемой игры.

Работа приложения должна удовлетворять сценарию использования, представленного в виде диаграммы (см. рис. 2.1).

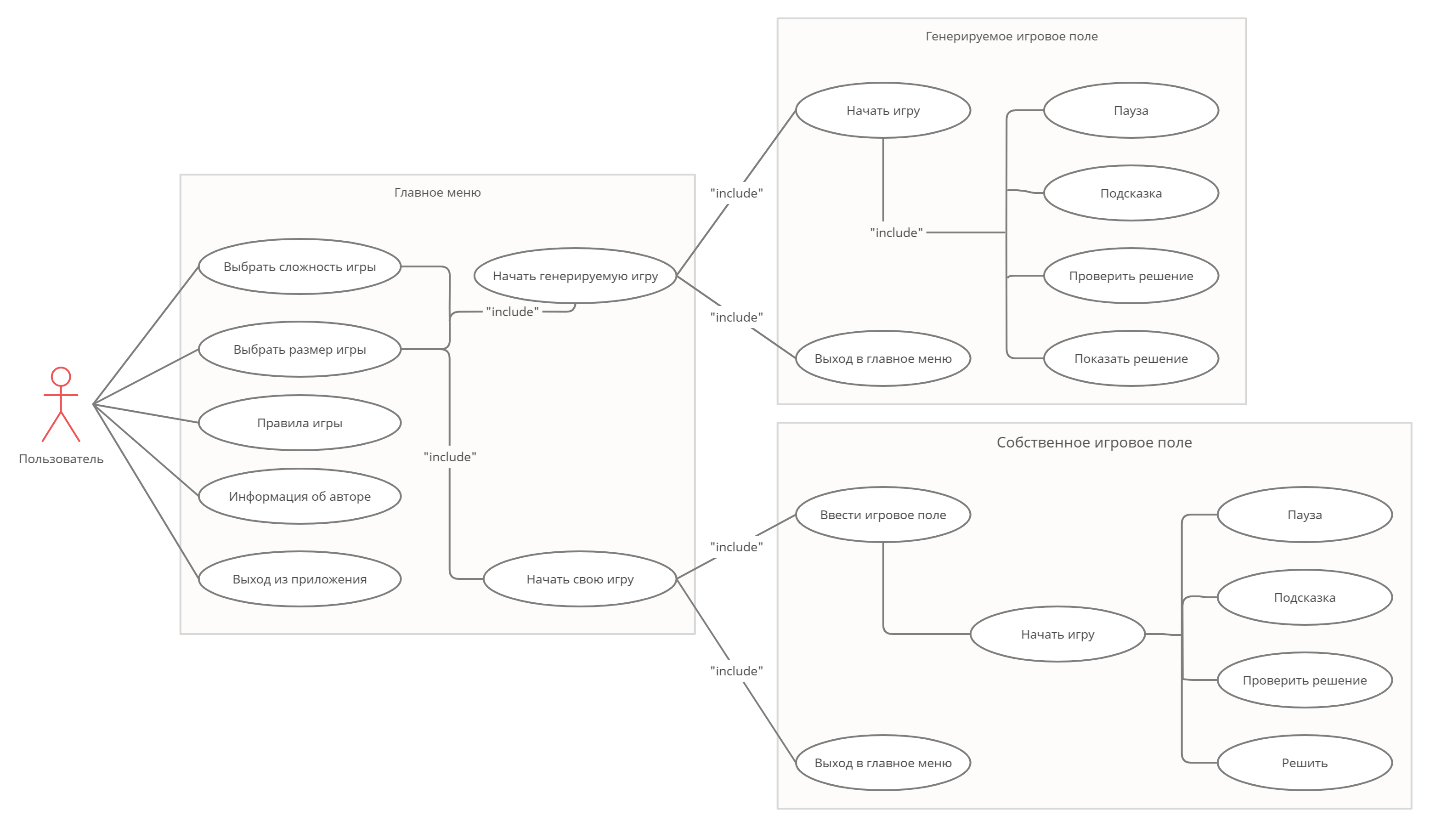


Рис. 2.1 – Диаграмма сценария использования приложения

## **Обоснование выбора языка программирования**

Для программной реализации игрового приложения Судоку были выбраны среды разработки Microsoft Visual Studio и JetBrains Rider и язык программирования C#.

Выбор данного языка обоснован его возможностями и сравнительной простотой в написании кода, а также уникальными средствами и инструментами, позволяющими структурировать и комментировать код.

Так как игра должна иметь пользовательский интерфейс, то было использовано приложение Windows Forms, открывающее практически безграничные возможности по созданию интерфейса, что, несомненно, является большим плюсом как для разработчика, так и для пользователя.

## **Программная реализация**

При разработке приложения программа была разделена на 6 классов:

* Sudoku – класс для хранения и обработки Судоку;
* GeneratedGame – класс для логики игры с генерацией поля;
* MyGame – класс для логики игры с вводом поля пользователем;
* StartForm – класс формы главного меню;
* GameForm – класс формы для генерируемой игры;
* MyGameForm – класс формы для собственной.

Они представлены в виде диаграммы классов, приведенной на рис. 2.3.1.

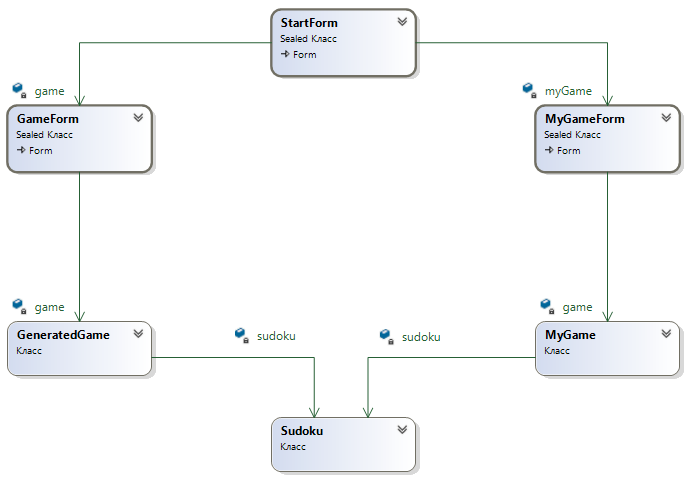


Рис. 2.3.1 – Диаграмма классов

Для понимания работы классов будет приведено описание полей и основных методов в каждом классе.

1. Sudoku

* *blocksize* – размер блока Судоку;
* sudokusize – размер Судоку;
* *map* – одномерный массив поля Судоку;
* *solve* – одномерный массив решенного поля Судоку;
* *cand* – лист кандидатов для каждой ячейки;
* *difficult* – сложность Судоку;
* *rand* – переменная класса Random для генерации поля;
* *GenerateMap()* – генерирует случайное поле для игры;
* *HideCells()* – скрывает ячейки исходя из уровня сложности, проверяя на решаемость Судоку при каждой итерации;
* *ShuffleMap(int i)* – выбирает один из четырех способов перемешивания Судоку;
* *SwapBlockInRow()* – меняет местами две случайных строки из блоков;
* *SwapBlockInColumn()* – меняет местами два случайных столбца из блоков;
* *SwapRowsInBlock()* – меняет местами строки в пределах блока;
* *SwapColumnsInBlock(*) – меняет местами колонки в пределах блока;
* *MatrixTransposition()* – транспонирует матрицу;
* *SolveByBruteForceWithReturn()* – решает Судоку полным перебором, заполняя поле *solve*;
* *BruteForceWithReturnRecursion(int index)* – рекурсивный алгоритм полного перебора, index – номер рассматриваемой ячейки;
* *SolveByModifiedBruteForceWithReturn()* – решает Судоку модифицированным полным перебором, заполняя поле *solve*;
* *ModifiedBruteForceWithReturnRecursion(int index)* – рекурсивный алгоритм модифицированного полного перебора;
* *IndexOfMinСandidates()* – ищет индекс ячейки с наименьшим числом кандидатов;
* *CountСandidates()* – считает количество кандидатов для каждой ячейки, заполняя поле *cand*;
* *CheckPossibleValue(int value, int index, int[] mySolution)* – проверяет валидное значение для ячейки матрицы, value – проверяемое значение, index – номер ячейки, *mySolution* – матрица, в которой идёт проверка;
* *CheckCorrectnessSudoku()* – проверяет Судоку на корректность заполнения;
* *CheckCorrectnessUserSudoku(int[] mySolution)* – проверяет введенное пользователем решение на корректность при своей игре в Судоку, *mySolution* – матрица решения;

1. GeneratedGame

* *blocksize* – размер блока Судоку;
* *sudoku* – экземпляр класса *Sudoku*;
* *GetMapNumber(int x, int y)* – возвращает число в ячейке массива *map*, *x* – строка, *y* - столбец;
* *GetSolveNumber(int x, int y)* – возвращает число в ячейке массива *solve*, *x* – строка, *y* - столбец;
* *StartGame()* – начинает игру;
* *CheckingGame(int[] mySolution)* – проверяет игру на выигрыш, *mySolution* – массив решения;

1. MyGame

* *blockSize* – размер блока Судоку;
* *sudoku* – экземпляр класса *Sudoku*;
* *GetMapNumber(int x, int y)* – возвращает число в ячейке массива *map*, *x* – строка, *y* - столбец;
* *GetSolveNumber(int x, int y)* – возвращает число в ячейке массива *solve*, *x* – строка, *y* - столбец;
* *CheckCorrectness()* – проверяет введенное поле на корректность;
* *CheckSolvability()* – проверяет введенное поле на решаемость;
* *CheckingGame(int[] mySolution)* – проверяет игру на выигрыш, *mySolution* – массив решения;

1. StartForm

* *difficulty* – сложность игры;
* *size* – размер игры;
* *game* – форма для генерируемой игры;
* *myGame* – форма для своей игры;
* *StartGameButton\_Click()* – происходит при нажатии на кнопку *startGameBtn*;
* *RuleButton\_Click()* – происходит при нажатии на кнопку *ruleBtn*;
* *AboutAuthorButton\_Click()* – происходит при нажатии на кнопку *aboutAuthorButton*;
* *YourBtn\_Click()* – происходит при нажатии на кнопку *yourBtn*;
* *SizeComboBox\_SelectedIndexChanged()* – происходит при изменении выбранного индекса в *sizeComboBox*;
* *DifficultyComboBox\_SelectedIndexChanged()* - происходит при изменении выбранного индекса в *difficultyComboBox*;
* *ExitButton\_Click()* – происходит при нажатии на кнопку *exitBtn*;
* *StartForm\_FormClosed()* – происходит после закрытии формы;

1. GameForm
   * *wmp* – музыкальный проигрыватель.
   * *game* – генерируемая игра;
   * *gameFlag* – флаг о состоянии игры;
   * *cell* – координаты нажатой кнопки;
   * *seconds*, *minutes*, *hours* – время таймера;
   * *gameSize* – размер игры (блока);
   * *difficult* – сложность игры;
   * *sizeButton* – размер кнопок игрового поля;
   * *buttons* – массив для удобного хранения игровых кнопок;
   * *mySolution* – массив текущего решения, введенного пользователем;
   * *ConvertDifficult()* – конвертирует сложность игры из индексного представления в количество скрываемых ячеек для данного размера игры;
   * *MyGameForm\_Load* - происходит при загрузке формы;
   * *CreateButtons()* – создает игровое поле из кнопок;
   * *StartButton\_Click()* – происходит при нажатии на кнопку startButton;
   * *StartGame()* – начинает игру;
   * *PrintGame()* – показывает игровое поле на форме;
   * *OnCellFocused()* – происходит при фокусировке на кнопке;
   * *OnCellOutFocus()* – происходит при снятии фокусировки на кнопке;
   * *OnCellKeyPressed()* – происходит при нажатии на кнопку;
   * *CheckButton\_Click()* – происходит при нажатии на кнопку chekButton;
   * *ShowWrongCell()* – показывает ячейки с ошибками;
   * *DeleteButtons()* – удаляет игровые кнопки;
   * *HelpButton\_Click()* – происходит при нажатии на кнопку helpButton;
   * *SolveButton\_Click()* – происходит при нажатии на кнопку solveButton;
   * *gameTimer\_Tick()* – происходит при каждом тике таймера gameTimer;
   * *pauseButton\_Click* – происходит при нажатии на кнопку pauseButton;
   * *HScrollBar\_Scroll* – происходит при скролле hScrollBar;
   * *ExitButton\_Click* – происходит при нажатии на кнопку exitButton;
   * *MyGameForm\_FormClosing* – происходит при закрытии формы;
   * *MyGameForm\_FormClosed* – происходит после закрытии формы;
2. MyGameForm
   * *wmp* – музыкальный проигрыватель.
   * *game* – своя игра;
   * *gameFlag* – флаг о состоянии игры;
   * *cell* – координаты нажатой кнопки;
   * *seconds*, *minutes*, *hours* – время таймера;
   * *gameSize* – размер игры (блока);
   * *sizeButton* – размер кнопок игрового поля;
   * *buttons* – массив для удобного хранения игровых кнопок;
   * *mySolution* – массив текущего решения, введенного пользователем;
   * *MyGameForm\_Load* - происходит при загрузке формы;
   * *CreateButtons()* – создает игровое поле из кнопок;
   * *StartButton\_Click()* – происходит при нажатии на кнопку startButton;
   * *StartGame()* – начинает игру;
   * *PrintGame()* – показывает игровое поле на форме;
   * *OnCellFocused()* – происходит при фокусировке на кнопке;
   * *OnCellOutFocus()* – происходит при снятии фокусировки на кнопке;
   * *OnCellKeyPressed()* – происходит при нажатии на кнопку;
   * *CheckButton\_Click()* – происходит при нажатии на кнопку chekButton;
   * *ShowWrongCell()* – показывает ячейки с ошибками;
   * *DeleteButtons()* – удаляет игровые кнопки;
   * *HelpButton\_Click()* – происходит при нажатии на кнопку helpButton;
   * *SolveButton\_Click()* – происходит при нажатии на кнопку solveButton;
   * *gameTimer\_Tick()* – происходит при каждом тике таймера gameTimer;
   * *pauseButton\_Click* – происходит при нажатии на кнопку pauseButton;
   * *HScrollBar\_Scroll* – происходит при скролле hScrollBar;
   * *ExitButton\_Click* – происходит при нажатии на кнопку exitButton;
   * *MyGameForm\_FormClosing* – происходит при закрытии формы;
   * *MyGameForm\_FormClosed* – происходит после закрытии формы.

В программе Судоку хранится в виде одномерного массива размером *N*×*N*.

При генерации игры заполняется массив *solve*, являющийся решением Судоку, далее скрывается *m* ячеек и заполняется массив map, в котором пустые ячейки хранятся в виде нулей. Игроку показывается матрица *map*. После ввода пользователем решения, оно сравнивается со сгенерированным решением, хранящемся в массиве *solve*.

При игре с ручным вводом поля, введенное игроком Судоку хранится в массиве *map*, а при нажатии на кнопку начала игры, данное поле проверяется на корректность и решается с помощью алгоритма ПВ, при этом заполняется массив решения *solve*. При проверке введенного решения оно сравнивается с решением, возвращенным алгоритмом, или просто проверяется на корректность (если введенное Судоку имело несколько решений).

## **Описание интерфейса и функций приложения**

Реализованное игровое приложение состоит из трёх форм: главное меню, генерируемая игра и игра с вводом своего Судоку.

В главном меню пользователю предлагается выбрать размер игры и её сложность (см. рис. 2.3.1). После выбора размера становится активна кнопка “Ввести своё Судоку” (только при 4×4 и 9×9), позволяющая начать игру в режиме решения собственного Судоку. Если выбрать любой размер и сложность, то активной становится кнопка “Начать игру”, которая запускает генерируемую игру с выбранным размером и сложностью. Также в меню игроку предлагается ознакомиться с правилами игры (кнопка “Правила”, см. рис 2.3.2) и информацией об авторе (кнопка “Об авторе”, см. рис. 2.3.3). Нажатие на кнопку “Выход” позволяет выйти из приложения.



Рис. 2.3.1 – Главное меню

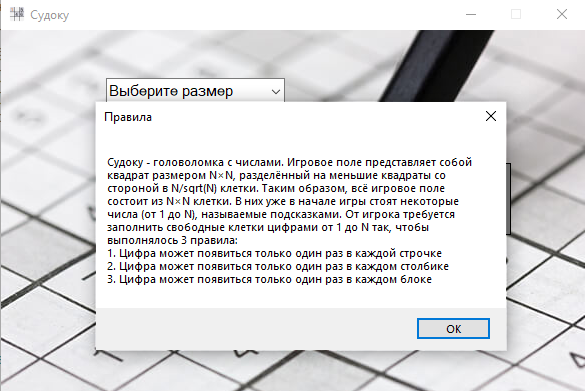


Рис. 2.3.2 – Информация о правилах игры

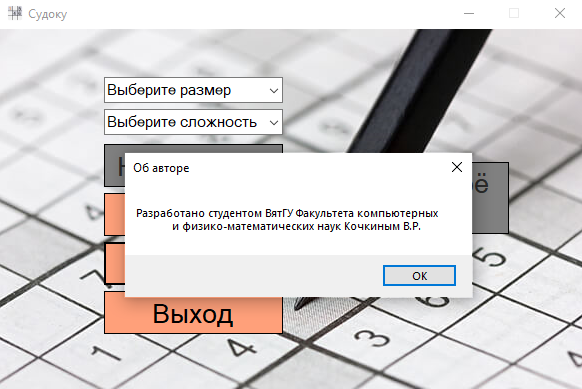


Рис. 2.3.3 – Информация об авторе

При нажатии на кнопку “Начать игру” в главном меню, открывается игровая форма (см. рис. 2.3.4), на которой расположены следующие элементы:

* Ползунок громкости – регулирует громкость музыки, выводя её процентное значение в текстовую строку.
* Информационная строка сложность – показывает текущую сложность игры.
* Таймер – показывает время игры.
* Кнопка “Начать игру” – запускает игру с выбранными в главном меню значениями размера и сложности.
* Кнопка “Пауза” – ставит игру на паузу, сменяется кнопкой “Продолжить”, которая продолжает игру. При паузе игровое поле скрывается. Пока игра не запущена, кнопка не доступна.
* Кнопка “Подсказка” – показывает верную цифру в выбранной ячейке.
* Кнопка “Проверить решение” – сверяет введенное решение со сгенерированным полем. Если поля не совпадают, то выводится сообщение о допущенных ошибках, и серым цветом выделяются ячейки с неверными значениями. Недоступна, пока игра не начата.
* Кнопка “Показать решение” – показывает сразу всё решение. Недоступна, пока игра не начата.
* Кнопка “Выход в главное меню” – производит выход в главное меню, но с предупреждением о незаконченной игре.

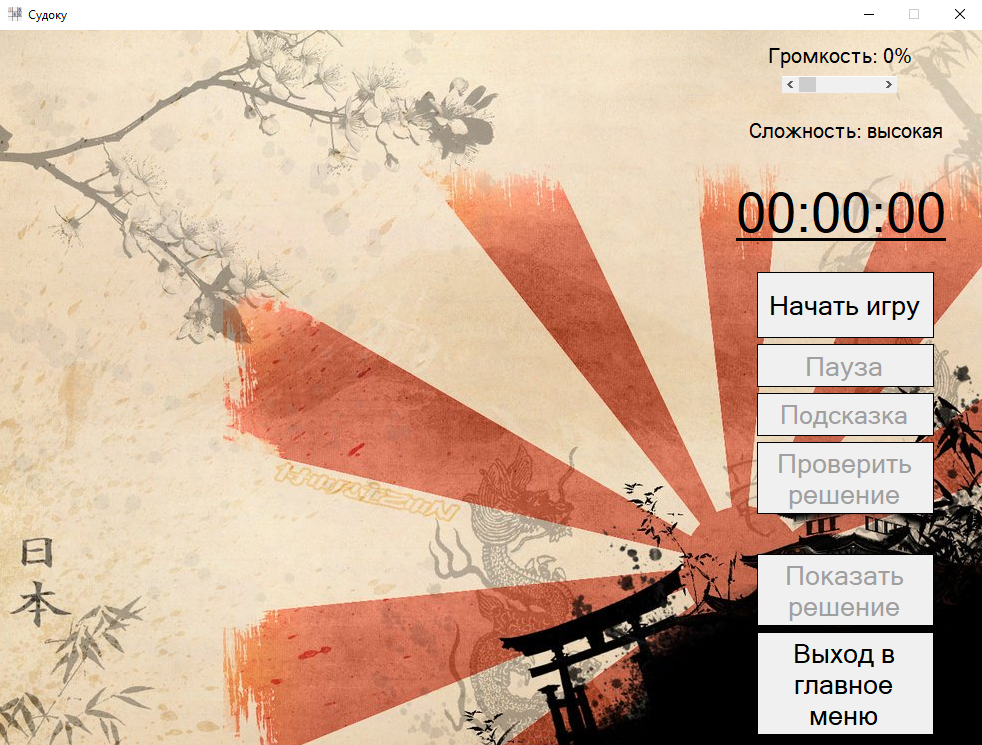
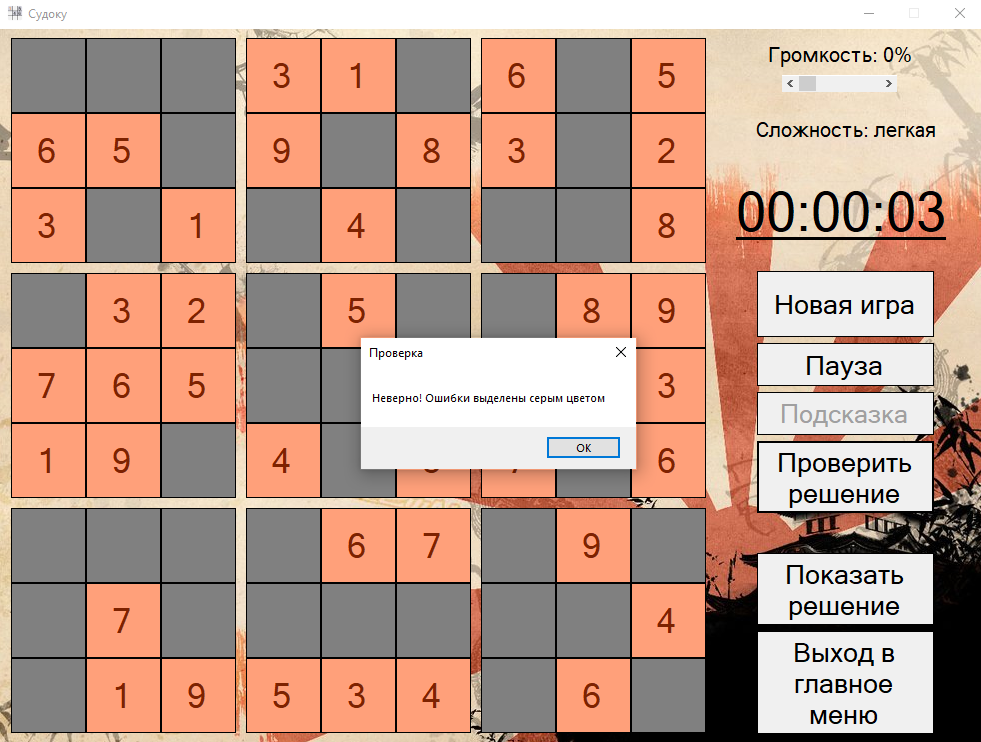


Рис. 2.3.4 – Интерфейс игровой формы

При нажатии на кнопку “Проверить решение”, если введенное решение не прошло проверку, показывается сообщение об ошибках и неверные ячейки перекрашиваются в серый цвет (см. рис. 2.3.5).



Если при активной игре пользователь нажмет кнопку “Выход в главное меню”, то программа выдаст предупреждение (см. рис. 2.3.5).

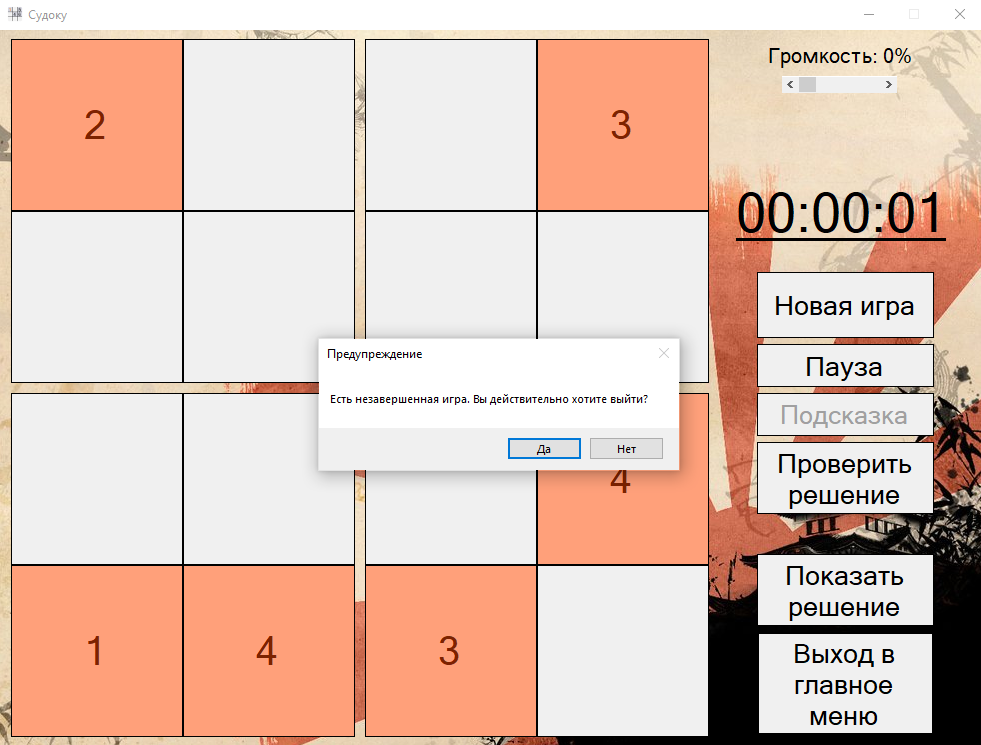


Рис. 2.3.5 – Выход в главное меню при активной игре

Если при активной игре пользователь нажмет кнопку “Новая игра”, то программа выдаст предупреждение о незаконченной партии (см. рис. 2.3.6). При нажатии на кнопку “Да” создается и генерируется новая игра.

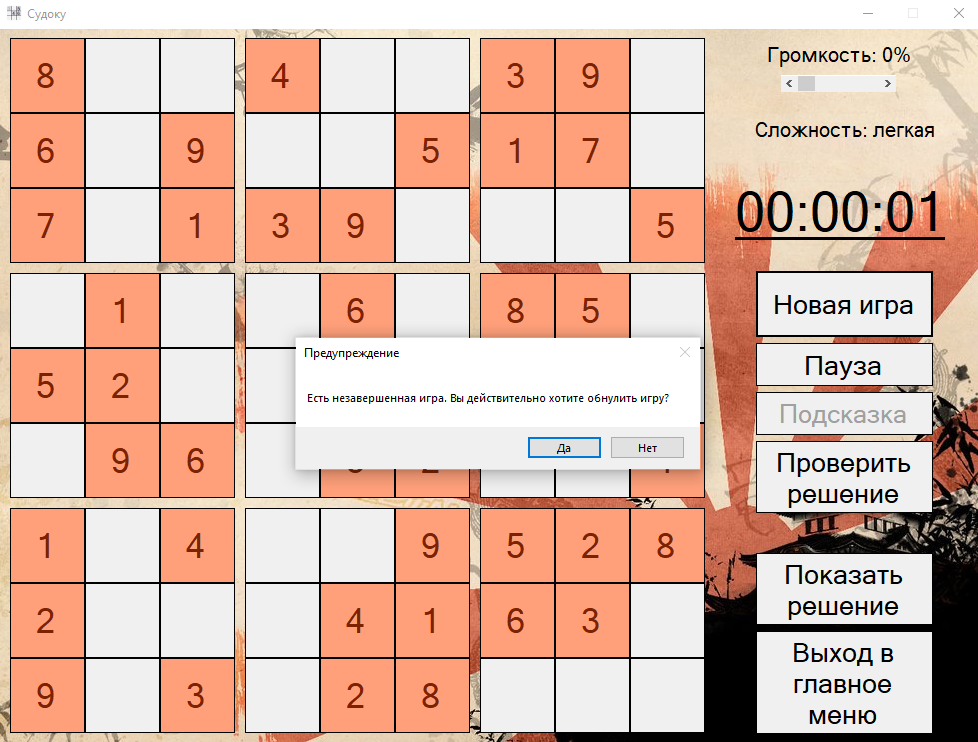


Рис. 2.3.6 – Нажатие на кнопку “Новая игра” при активной игре

Вторая игровая форма, предназначенная для ввода своего Судоку и дальнейшего его решения. Она имеет подобный интерфейс и функционал, как у первой игровой формы (см. рис. 2.3.5). Однако перед тем, как начать игру, необходимо ввести собственное Судоку в клеточки поля. Если поле не было введено, а пользователь нажал на кнопку “Начать игру”, то программа выдаст предупреждение (см. рис. 2.3.6). Также при старте игры, программа проверяет корректность введенного поля и его решаемость, о чем может выдать предупреждение (см. рис. 2.3.7).

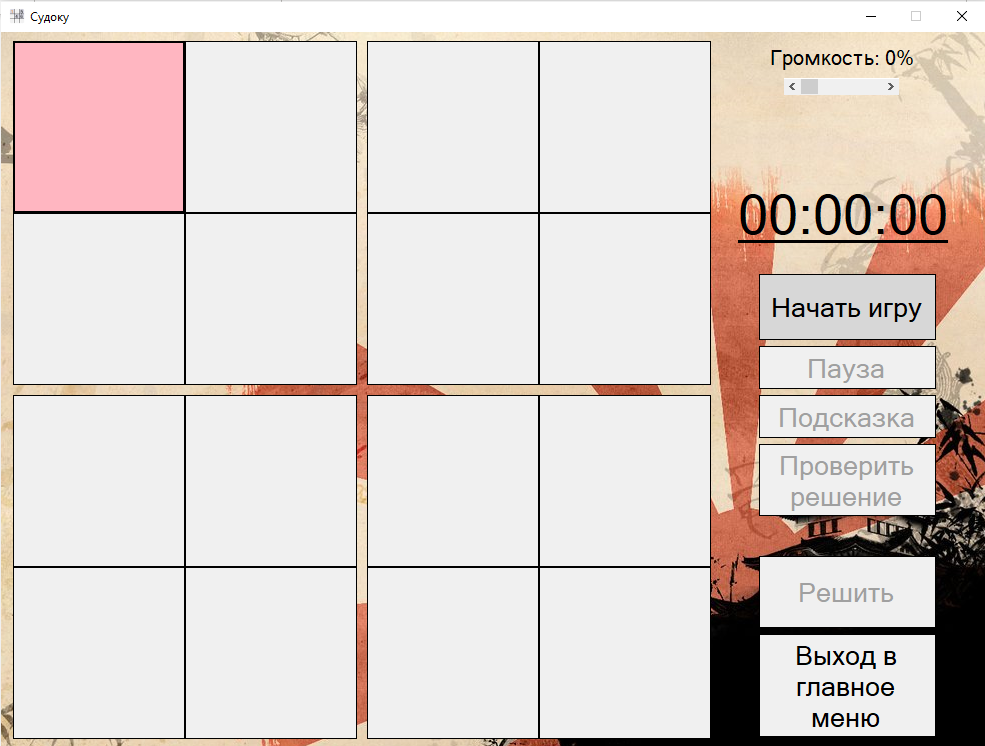


Рис. 2.3.5 – Игровая форма для решения своего Судоку

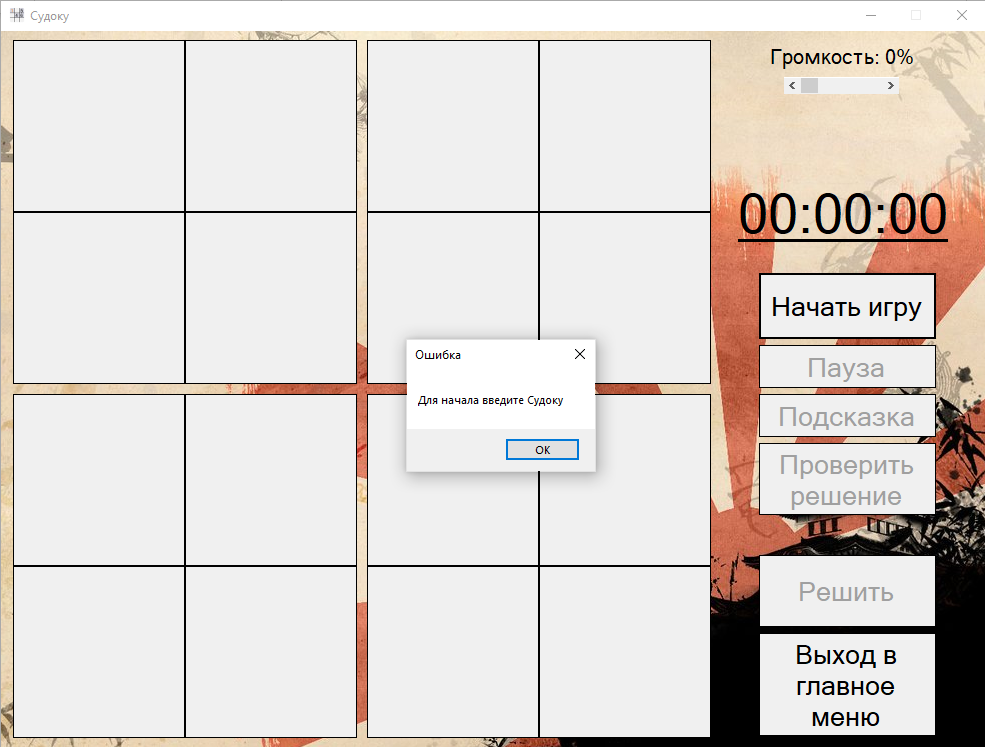


Рис. 2.3.6 – Обработка ситуации при старте игры с пустым полем

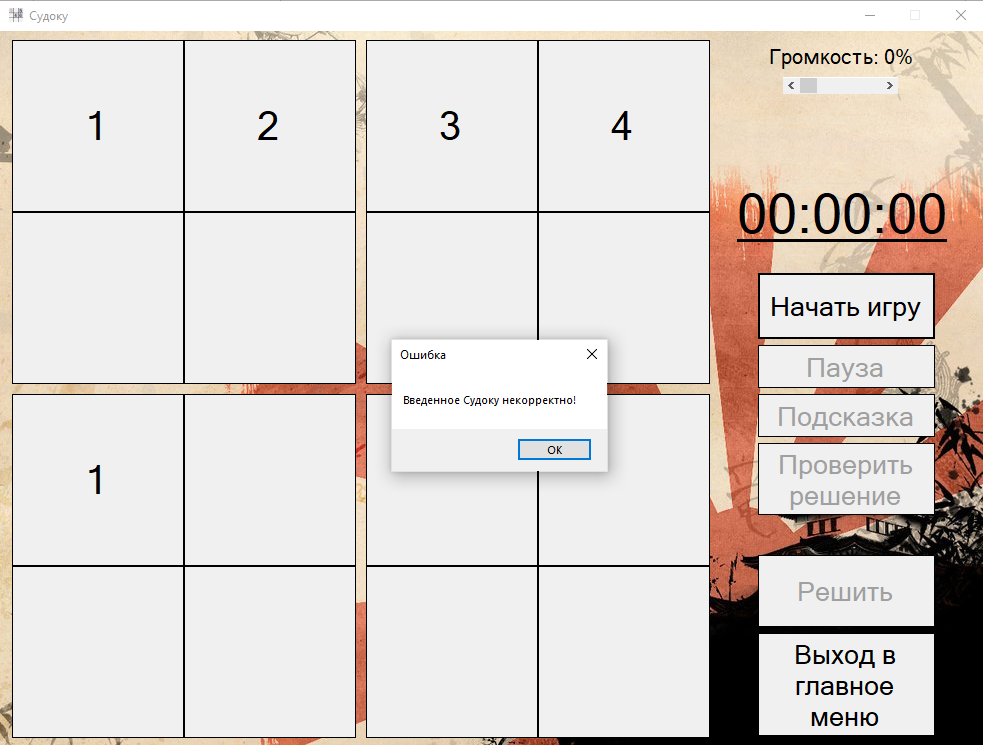


Рис. 2.3.7 – Обработка ситуации ввода некорректного Судоку

После начала игры введенные пользователем ячейки блокируются и запускается таймер.

## **Тестирование основных методов программы**

Для того чтобы удостовериться в правильности работы алгоритмов, были написаны тесты (UnitTest) основных методов программы.

Были протестированы алгоритмы решения Судоку, а также все их вспомогательные функции. В качестве исходных данных были взяты головоломки 4×4, 9×9 легкой, средней и высокой сложности. Для каждого Судоку было произведено тестирование, результаты которого представлены на рис. 2.4.

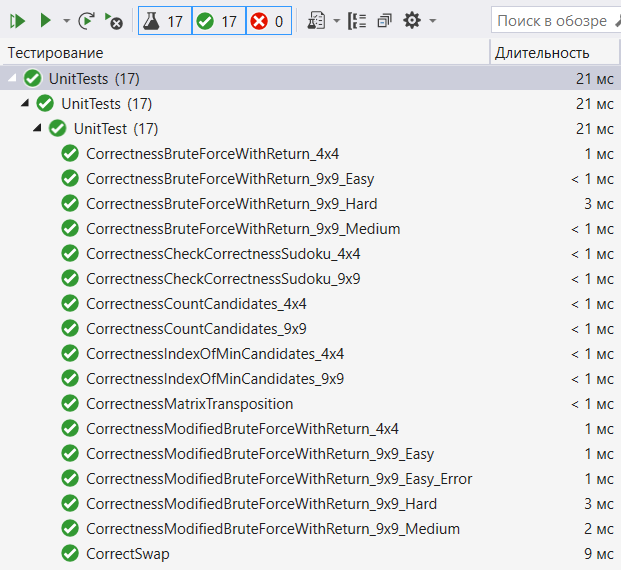


Рис. 2.4 – Результаты Unit тестов

В результате тестирования оба алгоритма правильно решали все данные им Судоку.

Листинг тестов можно найти в [Приложении Б](#_Приложение_Б._Листинг).

## **Выводы по разделу**

В разделе была описана практическая часть данной работы. Был проведен анализ требований к приложению, обоснован выбор языка программирования.

По данным требованиям было реализовано приложение и описана его программная составляющая, а также интерфейс. Проведены Unit-тесты, показывающие правильность работы основных методов программы, а также тесты работоспособности программы при некорректных действиях пользователя.

# **Заключение**

В данном курсовом проекте были решены все задачи и достигнута поставленная цель.

Во-первых, был произведен обзор аналогов игры Судоку и выявлен недостающий функционал, а именно отсутствие возможности ввода собственного Судоку и его решения.

Во-вторых, были рассмотрены алгоритм генерации и алгоритмы решения головоломки, а именно Алгоритм перебора с возвратом и его модификация. Они представлены в виде псевдокода и для наглядности продублированы примерами. Данные алгоритмы были протестированы на различных входных данных, благодаря чему сделаны выводы и выбран оптимальный алгоритм для реализации игрового приложения.

В-третьих, было спроектировано и разработано приложение Судоку, которое включает в себя следующие функции:

* Выбор режима игры: генерируемая игра или решение собственного Судоку.
* Выбор размера и сложности игры. Для генерируемой игры доступны размеры 4×4, 9×9, 16×16, 25×25 и сложности легкая, средняя, высокая и эксперт. Для собственной игры доступны размеры 4×4 и 9×9.
* Подсказки для обоих режимов.
* Возможность показать всё решение для обоих режимов.
* Пауза.
* Фоновая музыка, громкостью которой можно управлять из приложения.

В-четвертых, были обработаны все непредвиденные ситуации поведения пользователя при использовании приложения, а также написаны Unit-тесты, которые показывают корректность основных методов программы.

Таким образом, результатом курсового проекта стало игровое приложение Судоку, основанное на рассмотренных алгоритмах, которые были описаны благодаря анализу научной литературы.

# **Библиографический список**

1. Алгоритм генерации судоку. — Текст: электронный // Хабр. — URL: [https://habr.com/ru/post/192102/](https://habr.com/ru/post/192102/%20) (дата обращения 30.05.2021).
2. Банк Судоку [Электронный ресурс]. URL: <http://www.playsudoku.ru>.
3. Василенко С. Л. Числовая гармония судоку //режим доступа: <http://www.sciteclibrary.ru/texsts/rus/stat/st4689.pdf> – 2012.
4. Второй аналог приложения [Электронный ресурс]. URL: <https://sudoku.com/ru>.
5. Евстратов, В. В. Рекурсивный алгоритм решения судоку с проверкой найденного решения на единственность / В. В. Евстратов. — Текст: непосредственный // Молодой ученый. — 2020. — № 51 (341). — С. 8–11. — URL: [https://moluch.ru/archive/341/76568/](https://moluch.ru/archive/341/76568/%20) (дата обращения: 30.05.2021).
6. Кучма, М. Д. Алгоритмы решения головоломки "Судоку" / М. Д. Кучма // Заметки по информатике и математике: Сборник научных статей. – Ярославль: Ярославский государственный университет им. П. Г. Демидова, 2019. – С. 92–99.
7. Первый аналог приложения [Электронный ресурс]. URL: <https://sudoku-ru.com/>.
8. Пости, И. М. Об алгоритмах решения судоку на ЭВМ / И. М. Пости // Дни науки студентов Владимирского государственного университета имени Александра Григорьевича и Николая Григорьевича Столетовых: Сборник материалов научно-практических конференций, Владимир, 12 марта – 06 2018 года. – Владимир: Владимирский государственный университет им. Александра Григорьевича и Николая Григорьевича Столетовых, 2018. – С. 1118–1126.
9. Разновидности судоку: 21 способ развить свое мышление [Электронный ресурс]. URL: <https://neskuchayu.ru/puzzles/raznovidnosti-sudoku/>.
10. Третий аналог приложения [Электронный ресурс]. URL: <https://yandex.ru/games/play/97654?utm_campaign=main&utm_medium=yp&utm_source=Wizard&utm_term=standard>.

# **Приложения**

## **Приложение А. Листинг программы**

**Генерация Судоку.**

*/// <summary>  
/// Генерирует случайное поле  
/// </summary>*public void GenerateMap()  
{  
 Time.Restart();  
 for (var i = 0; i < sudokusize \* sudokusize; i++)  
 solve[i] = (i / sudokusize \* blocksize + i / sudokusize / blocksize + i % sudokusize) % sudokusize + 1;  
 var k = 0;  
 switch (blocksize)  
 {  
 case 2:  
 k = 100;  
 break;  
 case 3:  
 k = 1000;  
 break;  
 case 4:  
 k = 5000;  
 break;  
 case 5:  
 k = 10000;  
 break;  
 }  
 for (var i = 0; i < k; i++)  
 ShuffleMap(rand.Next(-1, 6));  
 Time.Stop();  
 map = (int[])solve.Clone();  
 HideCells();  
}  
  
*/// <summary>  
/// Скрывает ячейки  
/// </summary>*private void HideCells()  
{  
 var t = difficult;  
 while (t > 0)  
 {  
 for (var i = 0; i < sudokusize \* sudokusize; i++)  
 {  
 var a = rand.Next(0, 10);  
 if (a != 0 || map[i] == 0) continue;  
 map[i] = 0;  
 if (sudokusize == 2 || sudokusize == 3  
 ? SolveByBruteForceWithReturn()  
 : sudokusize != 4 || SolveByModifiedBruteForceWithReturn())  
 {  
 t--;  
 }  
 else map[i] = solve[i];  
 }  
 }  
}  
  
*/// <summary>  
/// Выбирает один из четырех способов преобразования матрицы  
/// </summary>  
/// <param name="i">Номер способа</param>*private void ShuffleMap(int i)  
{  
 switch (i)  
 {  
 case 0: MatrixTransposition(); break;  
 case 1: SwapRowsInBlock(); break;  
 case 2: SwapColumnsInBlock(); break;  
 case 3: SwapBlockInRow(); break;  
 case 4: SwapBlockInColumn(); break;  
 default: MatrixTransposition(); break;  
 }  
}  
  
*/// <summary>  
/// Меняет местами две ячейки массива  
/// </summary>  
/// <param name="a">Индекс первой ячейки</param>  
/// <param name="b">Индекс второй ячейки</param>*public void Swap(int a, int b)  
{  
 var temp = solve[a];  
 solve[a] = solve[b];  
 solve[b] = temp;  
}  
  
*/// <summary>  
/// Меняет местами две случайных строки из блоков  
/// </summary>*private void SwapBlockInRow()  
{  
 var block1 = rand.Next(0, blocksize);  
 var block2 = rand.Next(0, blocksize);  
 while (block1 == block2)  
 block2 = rand.Next(0, blocksize);  
 block1 \*= blocksize;  
 block2 \*= blocksize;  
 for (var i = 0; i < sudokusize; i++) *// i - столбец* {  
 var k = block2;  
 for (var j = block1; j < block1 + blocksize; j++)  
 {  
 Swap(i + j \* sudokusize, i + k \* sudokusize);  
 k++;  
 }  
 }  
}  
  
*/// <summary>  
/// Меняет местами два случайных столбца из блоков  
/// </summary>*private void SwapBlockInColumn()  
{  
 var block1 = rand.Next(0, blocksize);  
 var block2 = rand.Next(0, blocksize);  
 while (block1 == block2)  
 block2 = rand.Next(0, blocksize);  
 block1 \*= blocksize;  
 block2 \*= blocksize;  
 for (var i = 0; i < sudokusize; i++) *// i - строка* {  
 var k = block2;  
 for (var j = block1; j < block1 + blocksize; j++)  
 {  
 Swap(i \* sudokusize + j, i \* sudokusize + k);  
 k++;  
 }  
 }  
}  
  
*/// <summary>  
/// Меняет местами строки в пределах блока  
/// </summary>*private void SwapRowsInBlock()  
{  
 var block = rand.Next(0, blocksize);  
 var row1 = rand.Next(0, blocksize);  
 var line1 = block \* blocksize + row1;  
 var row2 = rand.Next(0, blocksize);  
 while (row1 == row2)  
 row2 = rand.Next(0, blocksize);  
 var line2 = block \* blocksize + row2;  
   
 for (var i = 0; i < sudokusize; i++)  
 {  
 Swap(line1 \* sudokusize + i, line2 \* sudokusize + i);  
 }  
}  
  
*/// <summary>  
/// Меняет местами колонки в пределах блока  
/// </summary>*private void SwapColumnsInBlock()  
{  
 var block = rand.Next(0, blocksize);  
 var col1 = rand.Next(0, blocksize);  
 var column1 = block \* blocksize + col1;  
 var col2 = rand.Next(0, blocksize);  
 while (col1 == col2)  
 col2 = rand.Next(0, blocksize);  
 var column2 = block \* blocksize + col2;  
 for (var i = 0; i < sudokusize; i++)  
 {  
 Swap(column1 + i \* sudokusize, column2 + i \* sudokusize);  
 }  
}  
  
*/// <summary>  
/// Транспониурет матрицу  
/// </summary>*public void MatrixTransposition()  
{  
 var tMap = new int[sudokusize \* sudokusize];  
 for (var i = 0; i < sudokusize; i++)  
 {  
 for (var j = 0; j < sudokusize; j++)  
 {  
 tMap[i \* sudokusize + j] = solve[j \* sudokusize + i];  
 }  
 }  
 solve = tMap;  
}

**Поиск решения методом ПВ.**

*/// <summary>  
/// Решает Судоку полным перебором  
/// </summary>  
/// <returns> Возвращает true, если решение было найдено, иначе - false</returns>*public bool SolveByBruteForceWithReturn()  
{  
 Time.Restart();  
 if (BruteForceWithReturnRecursion(0))  
 {  
 Time.Stop();  
 return true;  
 }  
 else return false;   
}  
  
*/// <summary>  
/// Рекурсивный алгоритм полного перебора  
/// </summary>  
/// <param name="index">Номер ячейки</param>  
/// <returns>Возвращает true, если решение было найдено, иначе - false</returns>*private bool BruteForceWithReturnRecursion(int index)  
{  
 switch (index < sudokusize \* sudokusize)  
 {  
 case true when solve[index] == 0:  
 {  
 for (var possibleValue = 1; possibleValue <= sudokusize; possibleValue++)  
 {  
 solve[index] = possibleValue;  
 if (CheckPossibleValue(possibleValue, index, solve) && BruteForceWithReturnRecursion(index + 1))  
 return true;  
 }  
 solve[index] = 0;  
 return false;  
 }  
 case true when !BruteForceWithReturnRecursion(index + 1):  
 return false;  
 default:  
 return true;  
 }  
}

**Поиск решения методом МПВ.**

*/// <summary>  
/// Решает Судоку модифицированным полным перебором  
/// </summary>  
/// <returns> Возвращает true, если решение было найдено, иначе - false</returns>*public bool SolveByModifiedBruteForceWithReturn()  
{  
 Time.Restart();  
 CountСandidates();  
 if (ModifiedBruteForceWithReturnRecursion(IndexOfMinСandidates()))  
 {  
 Time.Stop();  
 return true;  
 }  
 else return false;  
}  
  
*/// <summary>  
/// Рекурсивный алгоритм модифицированного полного перебора  
/// </summary>  
/// <param name="index">Номер ячейки</param>  
/// <returns>Возвращает true, если решение было найдено, иначе - false</returns>*private bool ModifiedBruteForceWithReturnRecursion(int index)  
{  
 if (solve.Contains(0))  
 {  
 var stackPossibleValues = new Stack<int>();  
 for (var possibleValue = 1; possibleValue <= sudokusize; possibleValue++)  
 {  
 if (CheckPossibleValue(possibleValue, index, solve))  
 stackPossibleValues.Push(possibleValue);  
 }  
 while (stackPossibleValues.Count > 0)  
 {  
 solve[index] = stackPossibleValues.Pop();  
 CountСandidates();  
 if (ModifiedBruteForceWithReturnRecursion(IndexOfMinСandidates()))  
 return true;  
 }  
 solve[index] = 0;  
 return false;  
 }  
 else return true;  
}  
  
*/// <summary>  
/// Ищет индекс ячейки с наименьшим числом кандидатов  
/// </summary>  
/// <returns>Возвращает индекс ячейки с наименьшим числом кандидатов</returns>*public int IndexOfMinСandidates()  
{  
 var min = sudokusize;  
 var index = 0;  
 for (var i = 0; i < cand.Count; i++)  
 if (solve[i] == 0 && cand[i] < min)  
 {  
 min = cand[i];  
 index = i;  
 }  
 return index;  
}  
  
*/// <summary>  
/// Считает количество кандидатов для каждой ячейки  
/// </summary>*public void CountСandidates()  
{  
 cand.Clear();  
 for (var i = 0; i < sudokusize \* sudokusize; i++)  
 {  
 cand.Add(0);  
 if (solve[i] != 0) continue;  
 for (var possibleValue = 1; possibleValue <= sudokusize; possibleValue++)  
 {  
 if (CheckPossibleValue(possibleValue, i, solve))  
 cand[i]++;  
 }  
 }  
}

**Проверка Судоку на корректность.**

*/// <summary>  
/// Проверяет валидное значение для ячейки матрицы  
/// </summary>  
/// <param name="value">Проверяемое значение</param>  
/// <param name="index">Номер ячейки</param>  
/// <param name="mySolution">Матрица, в которой проверяется значение</param>  
/// <returns>Возвращает true, если данное значение валидно, иначе - false</returns>*private bool CheckPossibleValue(int value, int index, int[] mySolution)  
{  
 *// Проверка по строкам и столбцам* for (var k = 0; k < sudokusize; k++)  
 {  
 if (index != index / sudokusize \* sudokusize + k &&  
 value == mySolution[(index / sudokusize) \* sudokusize + k])  
 return false;  
 if (index != index % sudokusize + sudokusize \* k &&  
 value == mySolution[(index % sudokusize) + sudokusize \* k])  
 return false;  
 }  
  
 *// Проверка по блокам* for (var k = 0; k < blocksize; k++)  
 {  
 for (var l = 0; l < blocksize; l++)  
 {  
 if (index != index / (sudokusize \* blocksize) \* sudokusize \* blocksize +  
 index % sudokusize / blocksize \* blocksize + sudokusize \* k + l  
 && value == mySolution[  
 index / (sudokusize \* blocksize) \* sudokusize \* blocksize +  
 index % sudokusize / blocksize \* blocksize + sudokusize \* k + l])  
 return false;  
 }  
 }  
 return true;  
}  
  
*/// <summary>  
/// Проверяет матрицу Судоку на корректность заполнения  
/// </summary>  
/// <returns>Возвращает true, если поле корректно, иначе - false</returns>*public bool CheckCorrectnessSudoku()  
{  
 for (var i = 0; i < sudokusize \* sudokusize; i++)  
 {  
 if (solve[i] == 0) continue;  
 if (!CheckPossibleValue(solve[i], i, solve))  
 return false;  
 }  
 return true;  
}  
  
*/// <summary>  
/// Проверяет решение пользователя на корректность заполнения  
/// </summary>  
/// <returns>Возвращает true, если поле корректно, иначе - false</returns>*public bool CheckCorrectnessUserSudoku(int[] mySolution)  
{  
 for (var i = 0; i < sudokusize \* sudokusize; i++)  
 {  
 if (!CheckPossibleValue(mySolution[i], i, mySolution))  
 return false;  
 }  
 return true;  
}

## **Приложение Б. Листинг Unit-тестов**

namespace UnitTests  
{  
 [TestClass]  
 public class UnitTest  
 {  
 [TestMethod]  
 public void CorrectSwap()  
 {  
 var field = new int[16]  
 {  
 1, 2, 3, 4,  
 4, 3, 2, 1,  
 3, 4, 1, 2,  
 2, 1, 4, 3  
 };  
 var sudoku = new Sudoku(2, field);  
 sudoku.Swap(0, 15);  
 Assert.AreEqual(sudoku.SolveMatrix[0], field[15]);  
 Assert.AreEqual(sudoku.SolveMatrix[15], field[0]);  
  
 sudoku.Swap(1, 1);  
 Assert.AreEqual(sudoku.SolveMatrix[1], field[1]);  
 }  
  
 [TestMethod]  
 public void CorrectnessCheckCorrectnessSudoku\_4x4()  
 {  
 var field = new int[16]  
 {  
 1, 0, 0, 4,  
 0, 3, 2, 0,  
 0, 4, 1, 0,  
 2, 0, 0, 3  
 };  
 var sudoku = new Sudoku(2, field);  
 Assert.AreEqual(true, sudoku.CheckCorrectnessSudoku());  
  
 var field1 = new int[16]  
 {  
 1, 0, 0, 4,  
 0, 2, 2, 0,  
 0, 4, 1, 0,  
 2, 0, 0, 3  
 };  
 var sudoku1 = new Sudoku(2, field1);  
 Assert.AreEqual(false, sudoku1.CheckCorrectnessSudoku());  
 }  
  
 [TestMethod]  
 public void CorrectnessCheckCorrectnessSudoku\_9x9()  
 {  
 var field = new int[81]  
 {  
 2, 0, 6, 9, 0, 7, 4, 0, 0,  
 5, 0, 0, 2, 8, 1, 7, 0, 0,  
 1, 9, 7, 0, 4, 0, 0, 0, 0,  
 7, 0, 0, 4, 0, 0, 0, 8, 0,  
 0, 0, 8, 0, 2, 6, 3, 7, 4,  
 0, 4, 0, 0, 7, 8, 0, 9, 2,  
 0, 6, 9, 3, 0, 0, 2, 4, 7,  
 4, 7, 0, 0, 9, 2, 0, 3, 1,  
 3, 2, 0, 7, 0, 0, 0, 0, 8  
 };  
 var sudoku = new Sudoku(3, field);  
 Assert.AreEqual(true, sudoku.CheckCorrectnessSudoku());  
  
 var field1 = new int[81]  
 {  
 2, 0, 6, 9, 0, 7, 4, 0, 0,  
 5, 0, 1, 2, 8, 1, 7, 0, 0,  
 1, 9, 7, 0, 4, 0, 0, 0, 0,  
 7, 0, 0, 4, 0, 0, 0, 8, 0,  
 0, 0, 8, 0, 2, 6, 3, 7, 4,  
 0, 4, 0, 0, 7, 8, 0, 9, 2,  
 0, 6, 9, 3, 0, 0, 2, 4, 7,  
 4, 7, 0, 0, 9, 2, 0, 3, 1,  
 3, 2, 0, 7, 0, 0, 0, 0, 8  
 };  
 var sudoku1 = new Sudoku(3, field1);  
 Assert.AreEqual(false, sudoku1.CheckCorrectnessSudoku());  
 }  
  
 [TestMethod]  
 public void CorrectnessMatrixTransposition()  
 {  
 var field1 = new int[16]  
 {  
 1, 2, 3, 4,  
 4, 3, 2, 1,  
 3, 4, 1, 2,  
 2, 1, 4, 3  
 };  
 var sudoku = new Sudoku(2, field1);  
 sudoku.MatrixTransposition();  
 var field2 = new int[16]  
 {  
 1, 4, 3, 2,  
 2, 3, 4, 1,  
 3, 2, 1, 4,  
 4, 1, 2, 3  
 };  
 for (int i = 0; i < field1.Length; i++)  
 {  
 Assert.AreEqual(field2[i], sudoku.SolveMatrix[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessBruteForceWithReturn\_4x4()  
 {  
 var field1 = new int[16]  
 {  
 4, 0, 0, 1,  
 0, 1, 3, 0,  
 0, 4, 1, 0,  
 1, 0, 0, 3  
 };  
 var sudoku = new Sudoku(2, field1);  
 Assert.AreEqual(true, sudoku.SolveByBruteForceWithReturn());  
 var field2 = new int[16]  
 {  
 4, 3, 2, 1,  
 2, 1, 3, 4,  
 3, 4, 1, 2,  
 1, 2, 4, 3  
 };  
 for (int i = 0; i < field1.Length; i++)  
 {  
 Assert.AreEqual(field2[i], sudoku.SolveMatrix[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessBruteForceWithReturn\_9x9\_Easy()  
 {  
 var field1 = new int[81]  
 {  
 2, 0, 6, 9, 0, 7, 4, 0, 0,  
 5, 0, 0, 2, 8, 1, 7, 0, 0,  
 1, 9, 7, 0, 4, 0, 0, 0, 0,  
 7, 0, 0, 4, 0, 0, 0, 8, 0,  
 0, 0, 8, 0, 2, 6, 3, 7, 4,  
 0, 4, 0, 0, 7, 8, 0, 9, 2,  
 0, 6, 9, 3, 0, 0, 2, 4, 7,  
 4, 7, 0, 0, 9, 2, 0, 3, 1,  
 3, 2, 0, 7, 0, 0, 0, 0, 8  
 };  
 var sudoku = new Sudoku(3, field1);  
 Assert.AreEqual(true, sudoku.SolveByBruteForceWithReturn());  
 var field2 = new int[81]  
 {  
 2, 8, 6, 9, 5, 7, 4, 1, 3,  
 5, 3, 4, 2, 8, 1, 7, 6, 9,  
 1, 9, 7, 6, 4, 3, 8, 2, 5,  
 7, 1, 2, 4, 3, 9, 5, 8, 6,  
 9, 5, 8, 1, 2, 6, 3, 7, 4,  
 6, 4, 3, 5, 7, 8, 1, 9, 2,  
 8, 6, 9, 3, 1, 5, 2, 4, 7,  
 4, 7, 5, 8, 9, 2, 6, 3, 1,  
 3, 2, 1, 7, 6, 4, 9, 5, 8  
 };  
 for (int i = 0; i < field1.Length; i++)  
 {  
 Assert.AreEqual(field2[i], sudoku.SolveMatrix[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessBruteForceWithReturn\_9x9\_Medium()  
 {  
 var field1 = new int[81]  
 {  
 2, 8, 0, 0, 4, 9, 0, 0, 3,  
 0, 0, 0, 0, 7, 6, 1, 0, 9,  
 0, 7, 0, 3, 0, 1, 0, 0, 0,  
 8, 5, 7, 0, 0, 0, 3, 0, 0,  
 4, 9, 0, 0, 0, 0, 0, 1, 2,  
 0, 0, 2, 0, 0, 0, 5, 6, 8,  
 0, 0, 0, 1, 0, 3, 0, 5, 0,  
 1, 0, 5, 7, 2, 0, 0, 0, 0,  
 7, 0, 0, 9, 5, 0, 0, 3, 1  
 };  
 var sudoku = new Sudoku(3, field1);  
 Assert.AreEqual(true, sudoku.SolveByBruteForceWithReturn());  
 var field2 = new int[81]  
 {  
 2, 8, 1, 5, 4, 9, 6, 7, 3,  
 5, 4, 3, 2, 7, 6, 1, 8, 9,  
 6, 7, 9, 3, 8, 1, 4, 2, 5,  
 8, 5, 7, 6, 1, 2, 3, 9, 4,  
 4, 9, 6, 8, 3, 5, 7, 1, 2,  
 3, 1, 2, 4, 9, 7, 5, 6, 8,  
 9, 2, 4, 1, 6, 3, 8, 5, 7,  
 1, 3, 5, 7, 2, 8, 9, 4, 6,  
 7, 6, 8, 9, 5, 4, 2, 3, 1  
 };  
 for (int i = 0; i < field1.Length; i++)  
 {  
 Assert.AreEqual(field2[i], sudoku.SolveMatrix[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessBruteForceWithReturn\_9x9\_Hard()  
 {  
 var field1 = new int[81]  
 {  
 0, 0, 0, 8, 0, 0, 4, 0, 5,  
 0, 0, 4, 0, 0, 5, 0, 9, 0,  
 0, 9, 0, 7, 0, 4, 0, 0, 1,  
 6, 0, 3, 0, 0, 8, 5, 1, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 5, 7, 9, 0, 0, 6, 0, 3,  
 1, 0, 0, 6, 0, 7, 0, 2, 0,  
 0, 7, 0, 1, 0, 0, 8, 0, 0,  
 5, 0, 9, 0, 0, 2, 0, 0, 0  
 };  
 var sudoku = new Sudoku(3, field1);  
 Assert.AreEqual(true, sudoku.SolveByBruteForceWithReturn());  
 var field2 = new int[81]  
 {  
 3, 2, 6, 8, 1, 9, 4, 7, 5,  
 7, 1, 4, 3, 6, 5, 2, 9, 8,  
 8, 9, 5, 7, 2, 4, 3, 6, 1,  
 6, 4, 3, 2, 7, 8, 5, 1, 9,  
 9, 8, 1, 5, 3, 6, 7, 4, 2,  
 2, 5, 7, 9, 4, 1, 6, 8, 3,  
 1, 3, 8, 6, 5, 7, 9, 2, 4,  
 4, 7, 2, 1, 9, 3, 8, 5, 6,  
 5, 6, 9, 4, 8, 2, 1, 3, 7  
 };  
 for (int i = 0; i < field1.Length; i++)  
 {  
 Assert.AreEqual(field2[i], sudoku.SolveMatrix[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessModifiedBruteForceWithReturn\_4x4()  
 {  
 var field1 = new int[16]  
 {  
 4, 0, 0, 1,  
 0, 1, 3, 0,  
 0, 4, 1, 0,  
 1, 0, 0, 3  
 };  
 var sudoku = new Sudoku(2, field1);  
 Assert.AreEqual(true, sudoku.SolveByModifiedBruteForceWithReturn());  
 var field2 = new int[16]  
 {  
 4, 3, 2, 1,  
 2, 1, 3, 4,  
 3, 4, 1, 2,  
 1, 2, 4, 3  
 };  
 for (int i = 0; i < field1.Length; i++)  
 {  
 Assert.AreEqual(field2[i], sudoku.SolveMatrix[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessModifiedBruteForceWithReturn\_9x9\_Easy\_Error()  
 {  
 var field1 = new int[81]  
 {  
 2, 0, 6, 9, 0, 7, 4, 0, 0,  
 5, 0, 0, 2, 8, 1, 7, 0, 0,  
 1, 9, 7, 0, 4, 0, 0, 0, 0,  
 7, 0, 0, 4, 0, 0, 0, 8, 0,  
 0, 0, 8, 0, 2, 6, 3, 7, 4,  
 0, 4, 0, 0, 7, 8, 0, 9, 2,  
 0, 6, 9, 3, 0, 0, 2, 4, 7,  
 4, 7, 0, 0, 9, 2, 0, 3, 1,  
 3, 2, 0, 7, 0, 0, 0, 0, 8  
 };  
 var sudoku = new Sudoku(3, field1);  
 Assert.AreEqual(true, sudoku.SolveByModifiedBruteForceWithReturn());  
 var field2 = new int[81]  
 {  
 2, 8, 6, 9, 5, 7, 4, 1, 3,  
 5, 3, 4, 2, 8, 1, 7, 6, 9,  
 1, 9, 7, 6, 4, 3, 8, 2, 5,  
 7, 1, 2, 4, 3, 9, 5, 8, 6,  
 9, 5, 8, 1, 2, 6, 3, 7, 4,  
 6, 4, 3, 5, 7, 8, 1, 9, 2,  
 8, 6, 9, 3, 1, 5, 2, 4, 7,  
 4, 7, 5, 8, 9, 2, 6, 3, 1,  
 3, 2, 1, 7, 6, 4, 9, 5, 8  
 };  
  
 var field3 = new int[81]  
 {  
 2, 8, 6, 9, 5, 7, 4, 1, 3,  
 5, 3, 4, 2, 8, 1, 7, 6, 9,  
 1, 9, 7, 6, 4, 3, 8, 2, 5,  
 7, 5, 2, 4, 3, 9, 1, 8, 6,  
 9, 1, 8, 5, 2, 6, 3, 7, 4,  
 6, 4, 3, 1, 7, 8, 5, 9, 2,  
 8, 6, 9, 3, 1, 5, 2, 4, 7,  
 4, 7, 5, 8, 9, 2, 6, 3, 1,  
 3, 2, 1, 7, 6, 4, 9, 5, 8  
 };  
  
   
 for (int i = 0; i < field1.Length; i++)  
 {  
 Assert.AreEqual(field3[i], sudoku.SolveMatrix[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessModifiedBruteForceWithReturn\_9x9\_Easy()  
 {  
 var field1 = new int[81]  
 {  
 0, 2, 7, 6, 0, 5, 3, 8, 0,  
 5, 0, 0, 7, 0, 8, 0, 0, 2,  
 1, 0, 8, 0, 0, 0, 6, 0, 5,  
 0, 1, 0, 2, 9, 3, 0, 4, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 4, 0, 5, 8, 6, 0, 9, 0,  
 8, 0, 2, 0, 0, 0, 1, 0, 7,  
 3, 0, 0, 8, 0, 7, 0, 0, 4,  
 0, 7, 4, 3, 0, 1, 9, 5, 0  
 };  
 var sudoku = new Sudoku(3, field1);  
 Assert.AreEqual(true, sudoku.SolveByModifiedBruteForceWithReturn());  
 var field2 = new int[81]  
 {  
 4, 2, 7, 6, 1, 5, 3, 8, 9,  
 5, 6, 9, 7, 3, 8, 4, 1, 2,  
 1, 3, 8, 9, 4, 2, 6, 7, 5,  
 7, 1, 5, 2, 9, 3, 8, 4, 6,  
 9, 8, 6, 1, 7, 4, 5, 2, 3,  
 2, 4, 3, 5, 8, 6, 7, 9, 1,  
 8, 5, 2, 4, 6, 9, 1, 3, 7,  
 3, 9, 1, 8, 5, 7, 2, 6, 4,  
 6, 7, 4, 3, 2, 1, 9, 5, 8  
 };  
 for (int i = 0; i < field1.Length; i++)  
 {  
 Assert.AreEqual(field2[i], sudoku.SolveMatrix[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessModifiedBruteForceWithReturn\_9x9\_Medium()  
 {  
 var field1 = new int[81]  
 {  
 2, 8, 0, 0, 4, 9, 0, 0, 3,  
 0, 0, 0, 0, 7, 6, 1, 0, 9,  
 0, 7, 0, 3, 0, 1, 0, 0, 0,  
 8, 5, 7, 0, 0, 0, 3, 0, 0,  
 4, 9, 0, 0, 0, 0, 0, 1, 2,  
 0, 0, 2, 0, 0, 0, 5, 6, 8,  
 0, 0, 0, 1, 0, 3, 0, 5, 0,  
 1, 0, 5, 7, 2, 0, 0, 0, 0,  
 7, 0, 0, 9, 5, 0, 0, 3, 1  
 };  
 var sudoku = new Sudoku(3, field1);  
 Assert.AreEqual(true, sudoku.SolveByModifiedBruteForceWithReturn());  
 var field2 = new int[81]  
 {  
 2, 8, 1, 5, 4, 9, 6, 7, 3,  
 5, 4, 3, 2, 7, 6, 1, 8, 9,  
 6, 7, 9, 3, 8, 1, 4, 2, 5,  
 8, 5, 7, 6, 1, 2, 3, 9, 4,  
 4, 9, 6, 8, 3, 5, 7, 1, 2,  
 3, 1, 2, 4, 9, 7, 5, 6, 8,  
 9, 2, 4, 1, 6, 3, 8, 5, 7,  
 1, 3, 5, 7, 2, 8, 9, 4, 6,  
 7, 6, 8, 9, 5, 4, 2, 3, 1  
 };  
 for (int i = 0; i < field1.Length; i++)  
 {  
 Assert.AreEqual(field2[i], sudoku.SolveMatrix[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessModifiedBruteForceWithReturn\_9x9\_Hard()  
 {  
 var field1 = new int[81]  
 {  
 0, 0, 0, 8, 0, 0, 4, 0, 5,  
 0, 0, 4, 0, 0, 5, 0, 9, 0,  
 0, 9, 0, 7, 0, 4, 0, 0, 1,  
 6, 0, 3, 0, 0, 8, 5, 1, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 5, 7, 9, 0, 0, 6, 0, 3,  
 1, 0, 0, 6, 0, 7, 0, 2, 0,  
 0, 7, 0, 1, 0, 0, 8, 0, 0,  
 5, 0, 9, 0, 0, 2, 0, 0, 0  
 };  
 var sudoku = new Sudoku(3, field1);  
 Assert.AreEqual(true, sudoku.SolveByModifiedBruteForceWithReturn());  
 var field2 = new int[81]  
 {  
 3, 2, 6, 8, 1, 9, 4, 7, 5,  
 7, 1, 4, 3, 6, 5, 2, 9, 8,  
 8, 9, 5, 7, 2, 4, 3, 6, 1,  
 6, 4, 3, 2, 7, 8, 5, 1, 9,  
 9, 8, 1, 5, 3, 6, 7, 4, 2,  
 2, 5, 7, 9, 4, 1, 6, 8, 3,  
 1, 3, 8, 6, 5, 7, 9, 2, 4,  
 4, 7, 2, 1, 9, 3, 8, 5, 6,  
 5, 6, 9, 4, 8, 2, 1, 3, 7  
 };  
 for (int i = 0; i < field1.Length; i++)  
 {  
 Assert.AreEqual(field2[i], sudoku.SolveMatrix[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessCountСandidates\_4x4()  
 {  
 var field = new int[16]  
 {  
 4, 0, 0, 1,  
 0, 1, 3, 0,  
 0, 4, 1, 0,  
 1, 0, 0, 3  
 };  
 var sudoku = new Sudoku(2, field);  
 sudoku.CountСandidates();  
 var cand = new int[16]  
 {  
 0, 2, 1, 0,  
 1, 0, 0, 2,  
 2, 0, 0, 1,  
 0, 1, 2, 0  
 };  
 for (int i = 0; i < cand.Length; i++)  
 {  
 Assert.AreEqual(cand[i], sudoku.Cand[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessCountСandidates\_9x9()  
 {  
 var field = new int[81]  
 {  
 0, 2, 7, 6, 0, 5, 3, 8, 0,  
 5, 0, 0, 7, 0, 8, 0, 0, 2,  
 1, 0, 8, 0, 0, 0, 6, 0, 5,  
 0, 1, 0, 2, 9, 3, 0, 4, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 4, 0, 5, 8, 6, 0, 9, 0,  
 8, 0, 2, 0, 0, 0, 1, 0, 7,  
 3, 0, 0, 8, 0, 7, 0, 0, 4,  
 0, 7, 4, 3, 0, 1, 9, 5, 0  
 };  
 var sudoku = new Sudoku(3, field);  
 sudoku.CountСandidates();  
 var cand = new int[81]  
 {  
 2, 0, 0, 0, 2, 0, 0, 0, 2,  
 0, 3, 3, 0, 3, 0, 1, 1, 0,  
 0, 2, 0, 2, 3, 3, 0, 1, 0,  
 2, 0, 2, 0, 0, 0, 3, 0, 2,  
 4, 5, 4, 2, 3, 1, 4, 5, 4,  
 2, 0, 1, 0, 0, 0, 2, 0, 2,  
 0, 3, 0, 2, 3, 2, 0, 2, 0,   
 0, 3, 4, 0, 3, 0, 1, 2, 0,  
 1, 0, 0, 0, 2, 0, 0, 0, 2  
 };  
 for (int i = 0; i < cand.Length; i++)  
 {  
 Assert.AreEqual(cand[i], sudoku.Cand[i]);  
 }  
 }  
  
 [TestMethod]  
 public void CorrectnessIndexOfMinСandidates\_4x4()  
 {  
 var field = new int[16]  
 {  
 4, 0, 0, 1,  
 0, 1, 3, 0,  
 0, 4, 1, 0,  
 1, 0, 0, 3  
 };  
 var sudoku = new Sudoku(2, field);  
 sudoku.CountСandidates();  
   
 Assert.AreEqual(2, sudoku.IndexOfMinСandidates());  
 }  
  
 [TestMethod]  
 public void CorrectnessIndexOfMinСandidates\_9x9()  
 {  
 var field = new int[81]  
 {  
 0, 2, 7, 6, 0, 5, 3, 8, 0,  
 5, 0, 0, 7, 0, 8, 0, 0, 2,  
 1, 0, 8, 0, 0, 0, 6, 0, 5,  
 0, 1, 0, 2, 9, 3, 0, 4, 0,  
 0, 0, 0, 0, 0, 0, 0, 0, 0,  
 0, 4, 0, 5, 8, 6, 0, 9, 0,  
 8, 0, 2, 0, 0, 0, 1, 0, 7,  
 3, 0, 0, 8, 0, 7, 0, 0, 4,  
 0, 7, 4, 3, 0, 1, 9, 5, 0  
 };  
 var sudoku = new Sudoku(3, field);  
 sudoku.CountСandidates();  
  
 Assert.AreEqual(15, sudoku.IndexOfMinСandidates());  
 }  
 }

}