МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

Рекурсия и головоломки

(наименование работы)

ОТЧЕТ

по лабораторной работе №4

по дисциплине

Технологии программирования

(наименование дисциплины)

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Капранов С.Н.

(подпись) (фамилия, и., о.)

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Русинова С.Д.

(подпись) (фамилия, и., о.)

18-ИСТ-4

(шифр группы)

Работа защищена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород, 2020

**ЗАДАНИЕ**

Реализовать игру Кен-кен.

Кен-кен – это математическая и логическая головоломка. Необходимо заполнить сетку цифрами так, чтобы в каждой строке и в каждом столбце они не повторялись. Число в углу каждого выделенного блока является результатом арифметической операции над цифрами в этом блоке. В отличие от судоку-убийцы (сум-до-ку), цифры внутри блока могут повторяться.

Реализованная игра должна иметь два режима: игрок может сам вводить ответ, либо программа будет сама решать задачу.

**Описание классов и функций**

* GameField – основная форма с игровым полем, генерирует регионы и правила, берет задание из файла:
* InitializeBoard – берет задание из файла, создает графическое отображение поля, генерирует области поля;
* CreateMap – создает графическую составляющую поля;
* GenerateRegions – разделяет поле на регионы, добавляет к регионам правила;
* MakeRule – генерация правила для данного региона, генерация значения региона, добавление его в список регионов;
* CellIsFree – определение состояния клетки;
* OccupyCell – обозначение функции как занятой;
* CompleteButton\_Click – проверка ответа на правильность;
* AutoCompleteButton\_Click – автоматическое решение задачи.
* KenKenBoard – класс игрового поля, определяет доступность клеток, значений, располагает их:
* CanPlace – определяет, можно ли расположить в данной точке данное число;
* UnsetPlace – очищает клетку;
* TryPlace – пробует расположить данное число на данной позиции.
* Region – класс региона, описывает характеристики одного выбранного региона:
* Prune – фильтрует решения для данного региона;
* IsValid – проверка значения на соответствие условиям задачи;
* AddExplore – перебирает подходящие комбинации в случае, если правило для региона – сложение;
* MultExplore – перебирает подходящие комбинации в случае, если правило для региона – умножение;
* GenerateSolutions – генерирует возможные решения в соответствии с оператором, добавляет их в список решений.
* Problem – класс задачи, отвечает за генерацию ответа:
* Prune – фильтрация подходящих ответов на уровне всей задачи, сохраняет их в список;
* Solve(KenKenBoard board)– для регионов доски просматривает все решения и пробует их применить, очищает доску, если решения не получились;
* Solve() – генерирует возможные решения для регионов, удаляет неподходящие, добавляет в список регионов.
* Cell – класс клетки, отвечает за координаты клетки, список кандидатов, ее состояние;
* Constants – класс, отвечающий за константы;
* EnumUtils – содержит обозначения различных операций для законов регионов (сумма, произведение и т.д.).

**Результат работы программы**

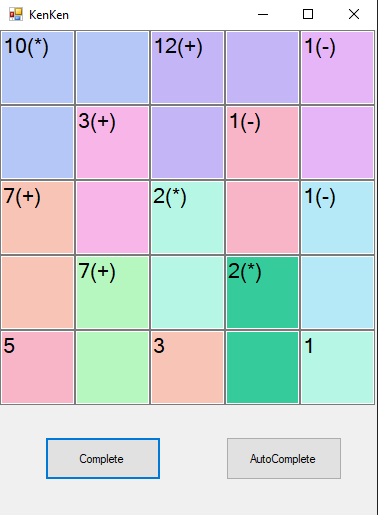


Рисунок - Вывод при запуске программыРисунок – Сообщение при вводе неверного ответа



Рисунок - Вывод при вводе правильного ответа

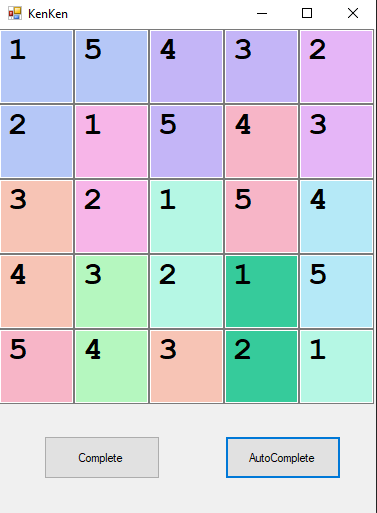


Рисунок - Вывод при автоматическом решении

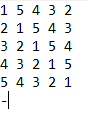


Рисунок - Исходный файл

**Листинг программы**

**GameField.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Windows.Forms;

using System.IO;

using static KENKENNN.EnumUtils;

using static KENKENNN.Constants;

using System.Linq;

using System.Runtime.InteropServices;

namespace KENKENNN

{

public partial class GameField : Form

{

// Экземпляр класса рандомизатора

private Random randomGenerator = new Random();

// Создаем массив текстбоксов

private TextBox[][] textCell = new TextBox[MapSize][];

// Создем массив, в который будем хранить информацию о том

// Задействованы ли клеточки в областях

private int[,] map = new int[MapSize, MapSize];

// Создаем массив с ответами

private int[,] correct = new int[MapSize, MapSize];

//Создаем новую задачу

private Problem problem;

[DllImport("kernel32.dll", SetLastError = true)]

[return: MarshalAs(UnmanagedType.Bool)]

static extern bool AllocConsole();

public GameField()

{

//Инифиализируем задачу

problem = new Problem();

InitializeComponent();

InitializeBoard();

AllocConsole();

}

#region GameCreation

private void InitializeBoard()

{

//Прозодимся по всем клеточкам поля

for (int i = 0; i < MapSize; i++)

{

for (int j = 0; j < MapSize; j++)

{

//Помечаем их как свободные

map[i, j] = FreeCell;

}

}

// Открытие и прочтение файла, на основе которого генерируется игра

var lines = File.ReadAllLines(@"../../Data/PuzzleData.txt");

var cells = new string[lines.Length][];

for (int i = 0; i < cells.Length; i++)

{

cells[i] = lines[i].Split(

new char[] { ' ', '\t' }, StringSplitOptions.RemoveEmptyEntries);

}

// Заполнение матрицы ответов

for (int i = 0; i < MapSize; i++)

{

for (int j = 0; j < MapSize; j++)

{

correct[i, j] = int.Parse(cells[i][j]);

}

}

//Создание отображения поля игры

CreateMap();

//Создание регионов

GenerateRegions();

}

// Функция создания поля игры

private void CreateMap()

{

// Задаем размер окна

Width = MapSize \* CellSize + CellSize / 2;

Height = MapSize \* CellSize + 2 \* CellSize;

// Создаем клеточки игрового поля

for (int i = 0; i < MapSize; i++)

{

textCell[i] = new TextBox[MapSize];

for (int j = 0; j < MapSize; j++)

{

textCell[i][j] = new TextBox();

textCell[i][j].Size = new Size(CellSize, CellSize);

textCell[i][j].Multiline = true;

textCell[i][j].Location = new Point(j \* CellSize, i \* CellSize);

textCell[i][j].Text = correct[i, j].ToString();

textCell[i][j].Font = new Font(textCell[i][j].Font.FontFamily, 16);

textCell[i][j].Text = string.Empty;

Controls.Add(textCell[i][j]);

}

}

// Задаем расположение кнопок

// CompleteButton.Location = new Point(0, Height - 2 \* CellSize);

// AutoComleteButton.Location = new Point(Width / 2, Height - 2 \* CellSize);

}

// Функиця генерации разделения игрового поля на области

private void GenerateRegions()

{

// Счетчик количества областей

var regionsCount = 0;

for (int i = 0; i < MapSize; i++)

{

for (int j = 0; j < MapSize; j++)

{

//Красим клетку

var cellColor = RegionColors[regionsCount];

//Создаем для данной клетки новый объект

var cell = new Cell(i, j);

//Новые объекты - соседние клетки, которые будут в регионе

Cell neighbourCcell1 = null, neighbourCell2 = null;

// Данное значение отвечает за основную клетку области

cell.Answer = correct[i, j];

// Если клетка не относится ни к одной из областей

if (CellIsFree(cell))

{

// Красим клетку в цвет, соответствующий региону

textCell[i][j].BackColor = cellColor;

//Помечаем клетку как занятую

OccupyCell(cell);

// Если клетка не на границе & клетка рядом свободна

if (i < MapSize - 1)

{

if (CellIsFree(i + 1, j))

{

neighbourCcell1 = new Cell(i + 1, j);

// Помечаем значение соседней клетки, записываем туда

// Правильный ответ в этой клетке

neighbourCcell1.Answer = correct[i + 1, j];

// Красим в цвет основной клетки

textCell[i + 1][j].BackColor = cellColor;

//Помечаем клетку как занятую

OccupyCell(neighbourCcell1);

}

}

// Если клетка не на границе & клетка рядом свободна

if (j < MapSize - 1)

{

if (CellIsFree(i, j + 1))

{

neighbourCell2 = new Cell(i, j + 1);

// Помечаем значение соседней клетки, записываем туда

// Правильный ответ в этой клетке

neighbourCell2.Answer = correct[i, j + 1];

// Красим в цвет основной клетки

textCell[i][j + 1].BackColor = cellColor;

//Помечаем клетку как занятую

OccupyCell(neighbourCell2);

}

}

//Создание правила для сформированной области

MakeRule(cell, neighbourCcell1, neighbourCell2);

regionsCount++;

}

}

}

}

// Создание правила для области игрового поля

// Сюда передаются значения основной и соседних клеточек, а так же их индексы

private void MakeRule(Cell primaryCell, Cell neighbourCell1, Cell neighbourCell2)

{

//Рандомно выбираем операцию

var regionOperation = (Operator)randomGenerator.Next((int)Operator.Add, (int)Operator.Div);

var operationValue = string.Empty;

//Основное значение региона

var regionValue = primaryCell.Answer;

// Если область состоит только из одной клеточки, то заполняем

// ее числом из ответа, как указано в правилах игры

if (CellIsFree(neighbourCell1) && CellIsFree(neighbourCell2))

{

regionOperation = Operator.Const;

}

else

{

switch (regionOperation)

{

//Если оператор +

case Operator.Add:

operationValue = "(+)";

if (CellIsFree(neighbourCell1))

{

regionValue += neighbourCell2.Answer;

}

else if (CellIsFree(neighbourCell2))

{

regionValue += neighbourCell1.Answer;

}

else

{

regionValue +=

(neighbourCell1.Answer + neighbourCell2.Answer);

}

break;

//Если оператор -

case Operator.Sub:

operationValue = "(-)";

int max, sum;

if (CellIsFree(neighbourCell1))

{

sum = primaryCell.Answer + neighbourCell2.Answer;

max = Math.Max(primaryCell.Answer, neighbourCell2.Answer);

regionValue = max - (sum - max);

}

else if (CellIsFree(neighbourCell2))

{

sum = primaryCell.Answer + neighbourCell1.Answer;

max = Math.Max(primaryCell.Answer, neighbourCell1.Answer);

regionValue = max - (sum - max);

}

else

{

sum = primaryCell.Answer + neighbourCell1.Answer + neighbourCell2.Answer;

{

regionOperation = Operator.Add;

operationValue = "(+)";

regionValue = sum;

}

}

break;

//Если оператор \*

case Operator.Mul:

operationValue = "(\*)";

if (CellIsFree(neighbourCell1))

{

regionValue \*= neighbourCell2.Answer;

}

else if (CellIsFree(neighbourCell2))

{

regionValue \*= neighbourCell1.Answer;

}

else

{

regionValue \*=

(neighbourCell1.Answer \* neighbourCell2.Answer);

}

break;

//Если оператор /

case Operator.Div:

operationValue = "(/)";

break;

default:

break;

}

}

//Заполняем значения основной клетки региона полученным

textCell[primaryCell.RowIndex][primaryCell.ColumnIndex].Text = regionValue.ToString() + operationValue.ToString();

//Создаем новый регион с данными значениями и оператором

var region = new Region(regionOperation, regionValue);

//Добавляем регион в список регионов

problem.Regions[primaryCell.RowIndex, primaryCell.ColumnIndex] = region;

if (neighbourCell1 != null)

{

//Помечаем клетку рядом, как входящую в регион

problem.Regions[neighbourCell1.RowIndex, neighbourCell1.ColumnIndex] = region;

}

if (neighbourCell2 != null)

{

//Помечаем клетку рядом, как входящую в регион

problem.Regions[neighbourCell2.RowIndex, neighbourCell2.ColumnIndex] = region;

}

}

//Функция определения состояния клетки

private bool CellIsFree(int rowIndex, int columnIndex)

{

return map[rowIndex, columnIndex] == FreeCell;

}

//Функция определения состояния клетки

private bool CellIsFree(Cell cell)

{

return cell == null ||

map[cell.RowIndex, cell.ColumnIndex] == FreeCell;

}

// Помечаем клетку как задействованную в области

private void OccupyCell(int rowIndex, int columnIndex)

{

map[rowIndex, columnIndex] = OccupiedCell;

}

private void OccupyCell(Cell cell)

{

map[cell.RowIndex, cell.ColumnIndex] = OccupiedCell;

}

#endregion

#region Buttons

private void Form1\_Load(object sender, EventArgs e) {}

// При нажатии на кнопку завершения

private void CompleteButton\_Click(object sender, EventArgs e)

{

int f = 0;

for (int i = 0; i < MapSize; i++)

{

for (int j = 0; j < MapSize; j++)

{

if (textCell[i][j].Text != correct[i, j].ToString())

{

f = 1;

break;

}

}

}

MessageBox.Show(f == 1 ? "Incorrect!" : "Correct!");

}

//Автоматическое решение

private void AutoComleteButton\_Click(object sender, EventArgs e)

{

var message = "Incorrect!";

var res = problem.Solve();

//Проходимся по всем решениям

if (res != null)

{

for (int rowIx = 0; rowIx < MapSize; rowIx++)

{

for (int colIx = 0; colIx < MapSize; colIx++)

{

var cell = textCell[rowIx][colIx];

//Заполняем клетки

cell.Text = res.State[rowIx, colIx].ToString();

cell.Font = new Font(FontFamily.GenericMonospace, 25, FontStyle.Bold);

}

}

// message = "Correct!";

}

// MessageBox.Show(message);

}

#endregion

}

}

**KenKenBoard.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using static KENKENNN.EnumUtils;

using System.Runtime.InteropServices;

namespace KENKENNN

{

//Класс объекта - доски

public class KenKenBoard

{

//Массив, в который будет записываться предполагаемый ответ

public int[,] State { get; private set; } = new int[Constants.MapSize, Constants.MapSize];

//Массивы отслеживают наличие чисел в строке/столбце

//Первый индекс - номер строки, второй - само число

bool[,] rowHash = new bool[Constants.MapSize, Constants.MapSize];

//Первый индекс - номер строки, второй - само число

bool[,] colHash = new bool[Constants.MapSize, Constants.MapSize];

//Метод, определяющий, можно ли расположить данное число на данных координатах

public bool CanPlace(Point p, int val)

{

//Необходимое для индексации преобразование

int valMinusOne = val - 1;

//Если данные ряды и стобцы уже заняты (=true), то расположить нельзя

return !rowHash[p.Y, valMinusOne] && !colHash[p.X, valMinusOne];

}

//Отменяет знятость сток/столбцов

public void UnsetPlace(Point p)

{

//Необходимое для индексации преобразование

int valMinusOne = State[p.X, p.Y] - 1;

//Теперь данные строки и столбцы свободны

rowHash[p.Y, valMinusOne] = false;

colHash[p.X, valMinusOne] = false;

State[p.X, p.Y] = 0;

}

//Попытка занять данным значением клетку

public bool TryPlace(Point p, int val)

{

//Необходимое для индексации преобразование

int valMinusOne = val - 1;

//Если априори нельзя поставить, то сразу выходим из функции

if (!CanPlace(p, val))

return false;

//Если можно, то помечаем строку и стобец как недоступные для данного числа

rowHash[p.Y, valMinusOne] = true;

colHash[p.X, valMinusOne] = true;

//Отмечаем в матрице ответов как занятое

State[p.X, p.Y] = val;

return true;

}

}

**Region.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using static KENKENNN.EnumUtils;

using System.Runtime.InteropServices;

//Класс объекта - игровой регион

public class Region

{

//Действие, которое выполняется на регионе (+,-..)

public Operator Operator { get; set; }

//Значение, полученное в ходе выполнения операции на регионе

public int Value { get; set; }

//Ячейки в регионе

public List<Point> Cells { get; private set; } = new List<Point>();

//Решения региона

public List<List<int>> Solutions { get; private set; }

public Region(Operator op, int val)

{

Operator = op;

Value = val;

}

//Фильтрация и уменьшение количества решений в соответствии с данным полем

public Region Prune(KenKenBoard board)

{

//Берем новый регион

var result = new Region(Operator, Value);

//Создаем для данного региона новый список решений

result.Solutions = new List<List<int>>();

//Создаем новые ячейки в данном регионе

result.Cells = Cells;

//Просмотр каждого решения

foreach (var solution in Solutions)

{

bool success = true;

//Проходимся по клеткам данного региона

for (int i = 0; i < Cells.Count; i++)

{

//Если хоть одну клетку решения нельзя расположить на доске

if (!board.CanPlace(Cells[i], solution[i]))

{

//То выходим из цикла, это решение не подошло

//Его не добавим к списку отфильтрованных

success = false;

break;

}

}

//Если получилось расположить, то добавялем в список

if (success)

result.Solutions.Add(solution);

}

//Если ни одно решение не подошло, то возвращаем null, иначе готовый решенный регион

return Solutions.Any() ? result : null;

}

//Проверка на правильность предлагаемого решения

public static bool IsValid(int val)

{

return val > 0 && val <= Constants.MapSize;

}

//Здесь происходит перебор всех возможных решений областей с законом - сложение

bool AddExplore(List<int> solutionCandidates, int currentVal)

{

//Данное значение всегда должно быть меньше значения области, так как в противном случае сумма не получится

if (currentVal > Value)

return true;

//Если список решений пуст, то создаем новый

if (solutionCandidates == null)

solutionCandidates = new List<int>();

//Если в списке решений на данный момент остается одно свободное место, ищем финальное число

if (solutionCandidates.Count == Cells.Count - 1)

{

//Т.к. ищем решения для суммы, вычитаем из значения региона данное значение

int val = Value - currentVal;

//Если полученное число может пристутствовать в решении, то

if (IsValid(val))

{

List<int> copy = new List<int>(solutionCandidates);

copy.Add(val);

//Добавляем в общий список решений данное

Solutions.Add(copy);

return true;

}

//Если числа, которые предлагается добавить к решению слишком большие, то вернется false

return val <= Constants.MapSize;

}

//Подставляем числа из доступных

for (int i = Constants.MapSize; i >= 1; i--)

{

//Сразу добавляем число в список

solutionCandidates.Add(i);

//Рекурсивно вызываем функцию

bool ret = AddExplore(solutionCandidates, currentVal + i);

//Предпоследнее полученное число убираем

solutionCandidates.RemoveAt(solutionCandidates.Count - 1);

//Прерываем просмотр потенциальных значений, так как они больше не будут подходить

if (!ret)

break;

}

return true;

}

//Здесь происходит перебор всех возможных решений областей с законом - сложение

bool MultExplore(List<int> solutionCandidates, int currentVal)

{

//Данное значение всегда должно быть меньше значения области, так как в противном случае произведение не получится

if (currentVal > Value)

return true;

//Если список решений пуст, то создаем новый

if (solutionCandidates == null)

solutionCandidates = new List<int>();

//Если в списке решений на данный момент остается одно свободное место, ищем финальное число

if (solutionCandidates.Count == Cells.Count - 1)

{

//Находим число как отношение, так как оператор - произведение

int val = Value / currentVal;

//Если полученное число может пристутствовать в решении, то

if (IsValid(val))

{

if (Value % currentVal != 0)

return true;

List<int> copy = new List<int>(solutionCandidates);

copy.Add(val);

//Добавляем в общий список решений данное

Solutions.Add(copy);

return true;

}

return val <= Constants.MapSize;

}

//Подставляем числа из доступных

for (int i = Constants.MapSize; i >= 1; i--)

{

//Сразу добавляем число в список

solutionCandidates.Add(i);

//Рекурсивно вызываем функцию

bool ret = MultExplore(solutionCandidates, currentVal \* i);

//Предпоследнее полученное число убираем

solutionCandidates.RemoveAt(solutionCandidates.Count - 1);

//Прерываем просмотр потенциальных значений, так как они больше не будут подходить

if (!ret)

break;

}

return true;

}

//Генерация возможных решений

public void GenerateSolutions()

{

//Если уже есть решения, больше создавать не надо, выходим

if (Solutions != null)

return;

//Объявляем список решений

Solutions = new List<List<int>>();

//Смотрим, какой закон у данного региона (+,-...)

switch (Operator)

{

//Если значения в данном регионе складываются

case Operator.Add:

//Рассматриваем все возможные решения области

AddExplore(null, 0);

break;

//Если значения в регионе вычитаются

case Operator.Sub:

//Для всех доступных значений

for (int a = 1; a <= Constants.MapSize; a++)

{

//Находим разность значения региона и данного

int b = Math.Abs(Value - a);

//Если получилось подходящее значение, то

if (IsValid(b))

{

//Добавляем эти значения в список ответов

Solutions.Add(new List<int> { a, b });

Solutions.Add(new List<int> { b, a });

}

}

break;

//Если значения в регионе делятся

case Operator.Div:

//Проходимся по всем доступным на поле значениям

for (int a = 1; a <= Constants.MapSize; a++)

{

//Находим отношение данного знаечния и значения региона

int b = a / Value;

//Если значение подходящее, то добавляем его в список решений

if (IsValid(b) &&

(a % Value == 0))

{

Solutions.Add(new List<int> { a, b });

Solutions.Add(new List<int> { b, a });

}

}

break;

//Если значения в регионе умножаются

case Operator.Mul:

//Перебором находим варианты решений

MultExplore(null, 1);

break;

case Operator.Const:

//Если регион состоит из одной клетки, то туда по правилам записываем уже имеющееся значение

Solutions.Add(new List<int> { Value });

break;

}

}

}

**Problem.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using static KENKENNN.EnumUtils;

using System.Runtime.InteropServices;

//Класс объекта - задание

public class Problem

{

//Задание состоит из списка регионов

List<Region> AllRegions = new List<Region>();

//Регион из регионов...?

public Region[,] Regions { get; } =

new Region[Constants.MapSize, Constants.MapSize];

//Фильтрация решений, которые больше не подходят для данной доски

//Принимает ситуацию на доске на данный момент

public Problem Prune(KenKenBoard board)

{

var result = new Problem();

//Для всех регионов

for (int i = 1; i < AllRegions.Count; i++)

{

//Объявляем данный регион

var region = AllRegions[i];

//Для данного региона отбрасываем недоходящие решения

var resultRegion = region.Prune(board);

//Если для данного региона не найдено решений

//То такой вариант решения не подошел

if (resultRegion == null)

return null;

//Иначе добавляем решение такого региона в список

result.AllRegions.Add(resultRegion);

}

return result;

}

//Эта функция решает задачу, когда найдены доступные решения, отсортированы, отфильтрованы

public KenKenBoard Solve(KenKenBoard board)

{

//Есть ли в списке регионы

if (!AllRegions.Any())

return board;

//Берем один регион

var region = AllRegions[0];

//Берем его клетки

var cells = region.Cells;

//Проходим через все возможные решения для данного региона

foreach (var solution in region.Solutions)

{

//Проходим по каждому отдельному предлагаемому значению

for (int i = 0; i < solution.Count; i++)

{

//Пробуем расположить данное значение на поле

board.TryPlace(cells[i], solution[i]);

}

//Отфильтруем доску с данным решением

var newProb = Prune(board);

//Если полученное решенное задание не пустое

if (newProb != null)

{

//Рекурсивно продолжаем искать решения для следующих областей

var ret = newProb.Solve(board);

if (ret != null)

return ret;

}

//Если вышло так, что алгоритм не смог найти подходящее решение

//То все, что осталось от него на доске, очищаем

foreach (Point p in cells)

board.UnsetPlace(p);

}

return null;

}

//Функция генерации решений, подготавливает почву для решения задачи

public KenKenBoard Solve()

{

//Проверка всех клеток на причастие к региону

for (int x = 0; x < Constants.MapSize; x++)

{

for (int y = 0; y < Constants.MapSize; y++)

{

if (Regions[x, y] != null)

//Если все ок, то побавляем данную клетку к списку клеток региона

Regions[x, y].Cells.Add(new Point(x, y));

}

}

//Генерируем решения для всех регионов

for (int x = 0; x < Constants.MapSize; x++)

{

for (int y = 0; y < Constants.MapSize; y++)

{

Regions[x, y].GenerateSolutions();

}

}

//Когда варианты ответов сгенерированы

//Создаем словарь регионов

var regions = new Dictionary<Region, int>();

//Временная переменная

Region tempSub;

//Проходимся по всему игровому полю

for (int x = 0; x < Constants.MapSize; x++)

{

for (int y = 0; y < Constants.MapSize; y++)

{

//Передаем во временную переменную регион на данных координатах

tempSub = Regions[x, y];

//Если такого региона нет в словаре

if (!regions.ContainsKey(tempSub))

{

//То доабвляем его

regions.Add(tempSub, tempSub.Cells.Count);

}

}

}

//Проходимся по списку регионов

foreach (var val in regions.Keys)

{

//Отбрасываем невозможные решения

//Для этого проходимся по всем возможным

foreach (var solution in new List<List<int>>(val.Solutions))

{

//Временная переменная для хранения игрового поля на данный момент

var tempBoard = new KenKenBoard();

//Смотрим все доступные решения

for (int i = 0; i < solution.Count; i++)

{

//Если при попытке расопложить такое значение в данном положении не вышло

if (!tempBoard.TryPlace(val.Cells[i], solution[i]))

{

//То такое решение удаляем и выходим. Оно точно не будет использоваться

val.Solutions.Remove(solution);

break;

}

}

}

//После фильтрации этого региона, добавляем его к списку регионов

AllRegions.Add(val);

}

//Пробуем решить задачу с полученными ресурсами, передаем пустое поле

return Solve(new KenKenBoard());

}

}

**Cell.cs**

using System.Collections.Generic;

namespace KENKENNN

{

public class Cell

{

//Координаты клетки

public int RowIndex { get; set; }

public int ColumnIndex { get; set; }

//Список потенциальных значений для данной клетки

public List<int> Candidates { get; set; }

// Пока клетка пустая, данные значение будет 'FreeCell'

public int Answer { get; set; } = Constants.FreeCell;

public Cell(int rowIx, int colIx)

{

RowIndex = rowIx;

ColumnIndex = colIx;

}

}

}

**Constants.cs**

using System.Collections.Generic;

using System.Drawing;

namespace KENKENNN

{

public static class Constants

{

// размер матрицы

public const int MapSize = 5;

// Размер одной клетки

public const int CellSize = 75;

//Помечалки состояния клетки

public const int FreeCell = -1;

public const int OccupiedCell = 1;

// Словарь с различными цветами

public static readonly Dictionary<int, Color> RegionColors = new Dictionary<int, Color>

{

{0, Color.FromArgb(181, 199, 247 )},

{1, Color.FromArgb(196, 181, 247 )},

{2, Color.FromArgb(229, 181, 247)},

{3, Color.FromArgb(247, 181, 232)},

{4, Color.FromArgb(247, 181, 199)},

{5, Color.FromArgb(247, 196, 181)},

{6, Color.FromArgb(181, 247, 228 )},

{7, Color.FromArgb(181, 233, 247)},

{8, Color.FromArgb(181, 247, 191)},

{9, Color.FromArgb(054, 203, 155)},

{10, Color.FromArgb(247, 181, 199)},

{11, Color.FromArgb(247, 196, 181)},

{12, Color.FromArgb(181, 247, 228 )},

{13, Color.FromArgb(181, 233, 247)},

};

}

}

**EnumUtils.cs**

namespace KENKENNN

{

public static class EnumUtils

{

public enum Operator : int

{

Add = 1,

Sub,

Mul,

Div,

Const

}

}

}