**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**ОДЕССКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ ИМЕНИ И. И. МЕЧНИКОВА**

**ИНСТИТУТ МАТЕМАТИКИ, ЭКОНОМИКИ И МЕХАНИКИ**

Курсовая работа

по курсу "Разработка программного обеспечения"

Студента 2-го курса

Специальности прикладная математика

Сидоренко Андрея

Преподаватель: Пенко В.Г.

Одесса 2015 г.

План работы:

1. Постановка задачи. Формулировка модели.

2. Введение ограничений.

3. Логическая составляющая Sudoku. Особенности реализации.

4. Графическая составляющая Sudoku. Особенности реализации.

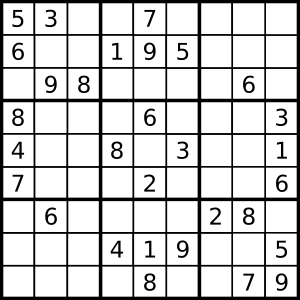
1. Постановка задачи

Sudoku — популярная головоломка с числами.

Правила:

Игровое поле представляет собой квадрат размером 9×9, разделённый на меньшие квадраты со стороной в 3 клетки. Таким образом, всё игровое поле состоит из 81 клетки. В них уже в начале игры стоят некоторые числа (от 1 до 9), называемые *подсказками*. От игрока требуется заполнить свободные клетки цифрами от 1 до 9 так, чтобы в каждой строке, в каждом столбце и в каждом малом квадрате 3×3 каждая цифра встречалась бы только один раз.

Пример:



**Цель:**

Перед нами стоит задача написать приложение для игры в Sudoku. Основные цели решения – предоставить удобный интерфейс для работы с приложением и фиксация всех логических ходов пользователя. Хотелось бы заметить, что перед нами не стоит задача решения заданного Sudoku. Таким образом мы должны в полной мере описать предметную область и перенести ее в графическую реализацию.

Математическая основа:

Сама же задача сводится к задаче о заполнении латинского квадрата. В обобщенном виде данная математическая проблема *не является однозначно решаемой*, что отображается в алгоритмах генерации Sudoku.

Пример латинского квадрата:


\begin{bmatrix}
A & B & C \\
C & A & B \\
B & C & A \\
\end{bmatrix}


Так же в теории доказано, что если подсказок 16 и меньше, то у судоку будет существовать несколько вариантов решений! Данное условия является необходимым, но не достаточным.

2. Введение ограничений. Описание алгоритма генерации.

В первую очередь нам нужно обусловить корректность задачи. В данном случае можно было выбрать два подхода генерации исходного Sudoku:

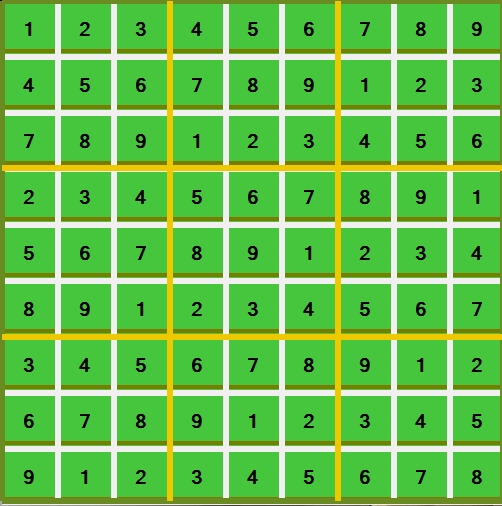
1. Методом полного перебора с обратным возвратом.

2. На основе начальной таблицы Sudoku, которое будет выступать семенем генерации всех последующих, создать произвольное Sudoku, которое будет иметь по крайней мере одно решение. *В таком случае алгоритм значительно упрощается*.

В решении данной задачи предлагаю действовать по 2-ой стратегии так как она проста в понимании и реализации. Таким образом опираясь на математическую базу и на алгоритм создания произвольного судоку мы не можем говорить о однозначности решения, но в силу того, что минимальное кол-во клеток-подсказок в нашей программе составляет 31 (уровень Hard – скрывает 50 клеток), можно утверждать, что большинство головоломок будут решаться однозначно. Данное ограничение является большим лимитирующим фактором нашей реализации судоку.

**Опишем алгоритм** создания произвольного судоку по используемой нами стратегии:

Перед нами стоит задача выбрать в качестве опорного элемента заполненную судоку, из общей теории построения латинских квадратов можно извлечь тривиальный случай вида:



(\*)

Элементы таблицы можно заполнить следующей формулой

ЭлементТаблицы[i, j] = (i \* 3 + i / 3 + j) % 9 + 1;

Рассмотрим операцию обмена значений val1 и val2 в под-сетке решенной судоку (\*):

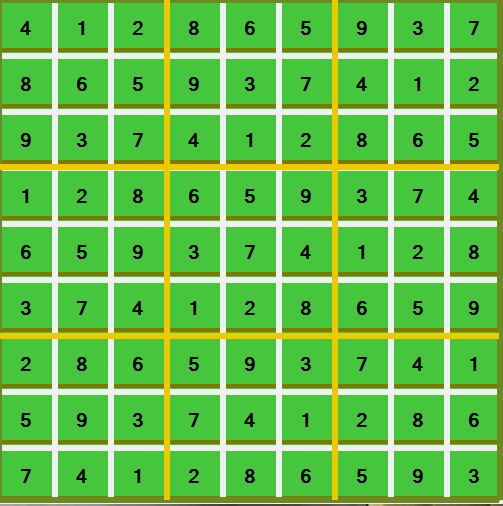
* В каждой строке (\*) содержится val1, val2
* В каждом столбце (\*) содержится val1, val2
* В каждой под-сетке (\*) содержится val1, val2

Утверждения выше очевидны.

Если в каждой под-сетке произвести обмен значениями val1 и val2, то 3-ие высказывание выполняется автоматически.

В силу того, что каждая строка/столбец содержит как val1 так и val2, которое в последующем будет заменено на val2 и val1 соответственно получаем, что состав каждой строки/столбца после обмена значениями не меняется. А значит условие корректности составленной судоку выполняется.

Проведя данную операцию замены n-ое кол-во раз получаем корректное судоку произвольного вида.



После чего скрываем заданное кол-во клеток в зависимости от указанного уровня сложности, сохранив при этом решенное судоку.

3. Написание логики Sudoku.

Логически разделим сетку на простейшие ячейки, которые будет хранить состояние клетки в Sudoku. Для этого напишем простой класс Cell(ячейка):

public class Cell

{

private int value; // значение

private bool isUnChangeable; // хранит информацию о способности клетки к изменению.

Данный класс является тривиальным и хранит несколько методов для работы с полями клетки.

Свойство CellVal – возвращает/присваивает значение в клетке.

Свойство IsUnchangeble – возвращает/присваивает статус клетки (описывается bool значением).

}

Sudoku состоит из 9-ти подтаблиц, в каждой из которых содержаться значения от 1 до 9 в единственном экземпляре. Так как в Sudoku некоторая часть логики направлена на работу с подтаблицами, напишем вспомогательный класс SubGrid(подсетка):

public class SubGrid

{

Cell[,] Arr = new Cell[3, 3];

…

Каждая под сетка содержит в себе 9 экземпляров класса Cell, которые образуют квадрат 3х3.

Помимо всех прочих методов для работы с полями класс содержит в себе 2 метода, которые можно отнести к особенностям реализации, опишем их подробнее:

1) Метод IsInSubGrid возвращает значение true, если заданное значение принадлежит под сетке. Есть две перегрузки, одна из которых возвращает строку и столбец заданного значения.

public bool IsInSubGrid(int val, out int posY, out int posX)

{

posY = -1; // Если такое значение не найдено, то по умолчанию возвращаем координату

posX = -1; // -1 ; -1

Далее следует полный перебор всех значений клеток в под сетке.

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

if (Arr[i, j].CellVal == val) // указанное значение найдено, подготавливаем возвращаемые значения.

{

posY = i;

posX = j;

return true;

}

}

}

return false;

}

2) Метод swapValue предназначен для корректной реализации генерации произвольного Sudoku. Меняет местами указанные значения в под сетке(если таковые имеются).

public void swapValue(int x, int y)

{

int xi, xj, yi, yj, tmpVal;

// находим позиции в под сетке указанных значений уже известным методом isInSubGrid

if (this.IsInSubGrid(x, out xi, out xj) && this.IsInSubGrid(y, out yi, out yj))

{

// После чего меняем их местами при помощи дополнительной переменной.

tmpVal = Arr[xi, xj].CellVal;

Arr[xi, xj].CellVal = Arr[yi, yj].CellVal;

Arr[yi, yj].CellVal = tmpVal;

}

}

Таким образом написав два простых метода мы описали поведение под сетки в Sudoku. Перейдем к написанию игровой логики используя написанные классы:

**Основной класс:**

public class SudokuGrid

{

Внутри класса SudokuGrid объявим перечисление обозначающие сложность составляемого Sudoku.

public enum GameDifficulty { Easy = 10, Medium = 25, Hard = 50 }

Массив 3х3 под сеток, которые в свою очередь содержат клетки судоку.

private SubGrid[,] Arr = new SubGrid[3, 3];

Поле ArrAnswer хранит в себе решение текущего Sudoku, сохраненное на этапе составления.

private SubGrid[,] ArrAnswer = new SubGrid[3, 3];

В данной задаче удобно работать как на уровне под-сеток, так и на уровне ячеек, поэтому напишем индексатор для быстрого доступа к ячейкам таблицы:

public Cell this[int y, int x]

{

//рассчитываем необходимый SubGrid, после чего обращаемся к его элементам Cell, через свойство GetCells и соответствующую координату.

get { return Arr[y / 3, x / 3].GetCells[y % 3, x % 3]; }

set { Arr[y / 3, x / 3].GetCells[y % 3, x % 3] = value; }

}

**#Генерация**

Нам следует реализовать алгоритм генерации судоку описанный выше.

Конструктор класса SudokuGrid:

public SudokuGrid(GameDifficulty diff = GameDifficulty.Easy)

{

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

Arr[i, j] = new SubGrid(); // заполняем незаполненными под сетками

}

Init(); // заполняем массив SubGrid’ов таблицей-ключом

Shuffle(); // “перетасовываем” таблицу судоку некоторое кол-во раз.

ArrAnswer = ((Sudoku.SubGrid[,])DeepClone(Arr)); // сохраняем ответ при помощи специального метода DeepClone

setDifficulty(diff); // скрываем некоторое кол-во клеток в зависимости от заданной сложности.

}

private void Init()

{

for (int i = 0; i < 9; i++)

{

for (int j = 0; j < 9; j++)

{

По описанному выше алгоритму заполняем generation-seed

this[i, j].CellVal = (i \* 3 + i / 3 + j) % 9 + 1;

} } }

private void Shuffle(int ShuffleDepth = 15)

{

// выполняем обмен значений указанное кол-во раз

if (ShuffleDepth-- > 0)

{

Random r = new Random(Guid.NewGuid().GetHashCode());

int val1, val2;

val1 = r.Next(1, 10);

val2 = r.Next(1, 10);

foreach (SubGrid sg in Arr)

{

Используем описанный выше метод в SubGrid – swapValue, которые меняет местами указанные значения в под-сетке.

sg.swapValue(val1, val2);

}

Продолжаем смешивать n-1 раз.

Shuffle(ShuffleDepth);

}

}

Метод setDifficulty скрывает указанное кол-во клеток.

**#Простейший ход в судоку**

В силу заданной композиции задача о заполнении судоку превращается в пару строк метода makeMove.

Замечание: Таблица судоку на логическом уровне заполняется только тогда, когда значение подходит под все 3 условия составления судоку. В противном случае заполняем значение по умолчанию - 0. Для того чтобы обеспечить переход от заполненного состояния в вакантное был написан метод makeWrongMove.

public bool makeMove(int y, int x, int val)

{

Условие принадлежности отрезку допустимых значений.

if (y >= 0 && y < 9 && x >= 0 && x < 9 && val < 10 && val > 0 )

{

Проверяем можно ли изменять значение в клетке

if (!this[y, x].IsUnchangable)

{

Проверяем выполнение 3-ех основных правил составления судоку.

Здесь пригодился выше описанный метод IsInSubGrid на уровне SubGrid для проверки значений входящих в под-сетку.

Методы checkRow и checkColumn проверяют входил ли указанное значение в строку/столбец и возвращают true если его там нет.

if (checkColumn(y, val) && checkRow(x, val) && !Arr[y / 3, x / 3].IsInSubGrid(val))

{

this[y, x].CellVal = val;

return true;

}

}

}

return false;

}

public void makeWrongMove(int y, int x)

{

if (y >= 0 && y < 9 && x >= 0 && x < 9)

{ Изменим значение изменяемой клетки на нулевое.

if (!this[y, x].IsUnchangable)

{

this[y, x].CellVal = 0;

}

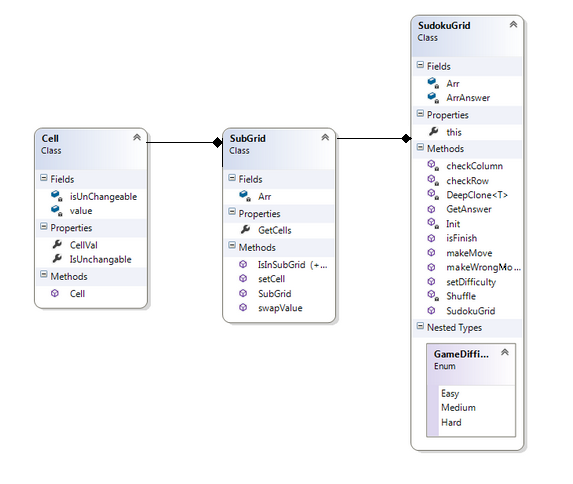
}

}

**Вывод:**

Таким образом разложив предметную область на структурные составляющие, которые несут за собой определенную логику, мы смогли достаточно просто сгенерировать судоку и обеспечить процесс игры в нее.

Диаграмма классов. Взаимодействие классов обеспечивается путем композиции.



4. Некоторые особенности графической реализации

* Связь графики и логики.

Класс Form1 содержит поле типа SudokuGrid, которое хранит логическое представление судоку описанное выше. Нашей задачей является передать логическую часть в графическую реализацию классом SudokuTableGraphic, который наследует графический элемент TableLayoutPanel, что позволяет обращаться с ним как с таблицей. Так же внутри класса есть ряд методов которые рисуют таблицу.

* Связь с пользователем, элементы управления

В графическую таблицу мы добавляем составной элемент класса SudokuCell класса, который наследует класс Panel. Это позволяет нам добавить в класс SudokuCell графический элемент класса numericUpDown и Label. (Все numericUpDown подписаны на 1 обработчик событий)

Роль numericUpDown получить значение набираемое пользователем. Обработчик событий

void numeric\_ValueChanged при изменении значения numericUpDown обрабатывает вводимое значение и в зависимости от входной таблицы SudokuGrid в случае успешного заполнения графически отображает в Label, все это происходит на уровне SudokuCell. Таким образом SudokuTableGraphic содержащий массив SudokuCell обрабатываем и изменяет таблицу в форме.

* Элементы сериализации

В силу способа отображения на форме при использовании сериализации понадобилась дополнительная информация о сериализуемых объектах, поэтому был написан класс MoveList с некоторым функционалом для хранения последних изменений клеток в таблице.

По смыслу, ход игрока – это отображение на игровую таблицу, поэтому использовали встроенный параметризуемый класс Dictionary<Key, Value>.

Dictionary<Point,int> moveLog: ключем выступает встроенный класс Point – координата в таблице и значением выступает значение в клетке. Таким образом кроме сериализации таблицы нам понадобится сериализовать класс MoveList. Разные типы сериализуемых данных сохраняем в разные файлы.

Сериализуем при вызове события void Form1\_FormClosed и десериализуем - Form1\_Load.