|  |  |
| --- | --- |
| Gerb-BMSTU_01**f** | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ Робототехники и комплексной автоматизации

КАФЕДРА Системы автоматизированного проектирования (РК-6)

**ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ**

по дисциплине: «Интеллектуальные системы»

|  |  |  |
| --- | --- | --- |
| Студент |  | Макаров Тимофей Геннадьевич |
| Группа |  | РК6-22М |
| Тип задания |  | Лабораторная работа |
| Тема |  | Алгоритмы локального поиска |

Студент **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_Макаров Т.Г.**

*подпись, дата фамилия, и.о.*

Преподаватель **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_ Божко А.Н.**

*подпись, дата фамилия, и.о.*

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Москва, 2023 г.*

Оглавление

[Оглавление 2](#_Toc134639966)

[Описание задачи 3](#_Toc134639967)

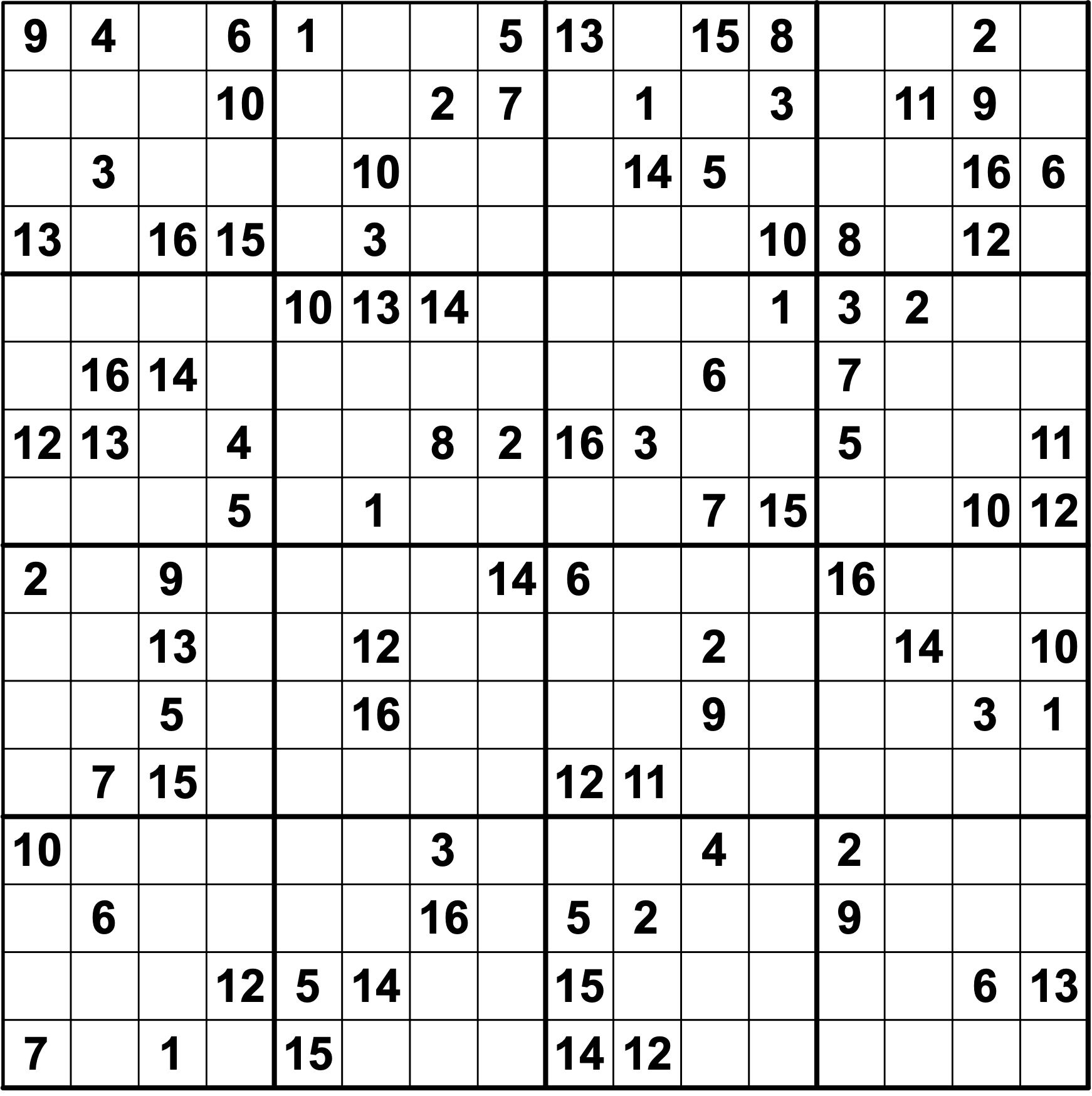
[Constraint Satisfaction Problem 3](#_Toc134639968)

[Результат работы программы 5](#_Toc134639969)

[Приложение 7](#_Toc134639970)

Описание задачи

Вариант 5



Требуется написать программу для решения судоку на любом языке программирования по заданному алгоритму.

Алгоритм: локальный жадный поиск с переменным соседством.

Описание алгоритма

К жадным алгоритмам относится любой алгоритм, который следует эвристике выбора локально оптимального шага на каждом этапе решения задачи, допуская, что конечное решение будет оптимальным.

Решение судоку алгоритмом жадного поиска с переменным соседством реализовано следующим образом.

1. Все пустые клетки судоку заполняются псевдослучайным образом так, чтобы в пределах малых квадратов числа не повторялись.
2. Оценивается суммарная ошибка полученной комбинации чисел в клетках (сумма числа повторяющихся значений в строках и столбцах).
3. Производится улучшение с постепенным увеличением числа генерируемых соседей. Генерация соседей происходит следующим образом. Малые квадраты нумеруются построчно, как показано на рисунке 2.

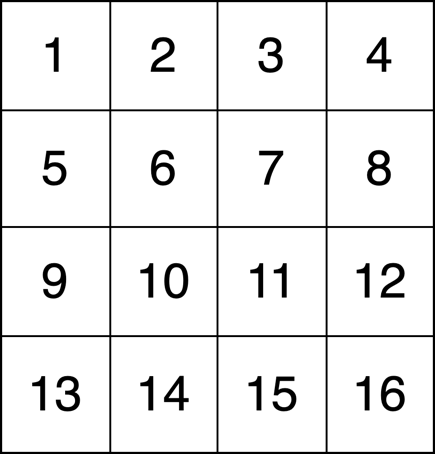


Рисунок 2 – Пример построчной нумерации малых квадратов для судоку 16х16.

Каждый квадрат представляется в виде одномерного массива. Сначала соседи генерируются путём циклического сдвига на 1 вправо в пределах первого квадрата части элементов (рисунок 3), которые не были заполнены в начальном условии. В случае, если удалось улучшить значение целевой функции, продолжаем генерацию тем же образом. В противном случае начинаем рассматривать дополнительно к первому квадрату второй. Продолжаем до тех пор, пока перестановки во всех квадратах не перестанут улучшать целевую функцию.

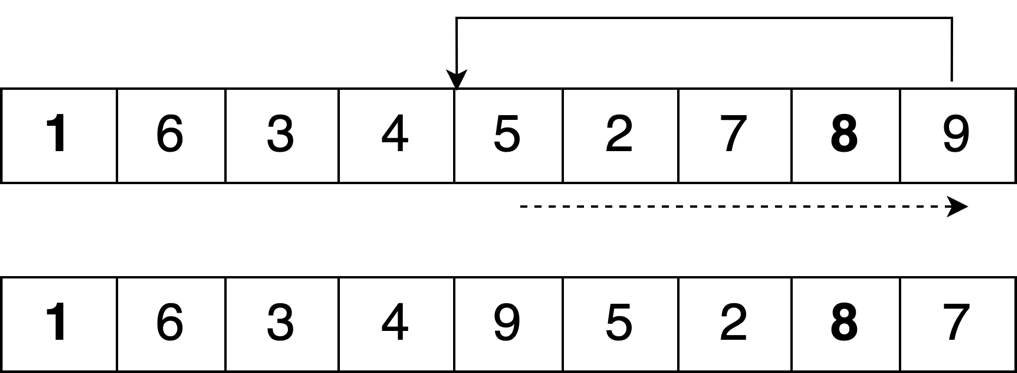


Рисунок 3 – Пример циклического сдвига блока в право. Жирным обозначены фиксированные элементы.

Переходим на другой метод генерации соседей, при котором производится обмен местами двух элементов в пределах одного квадрата (рисунок 4). При этом, в отличие от предыдущего метода, рассматриваются сразу все квадраты.

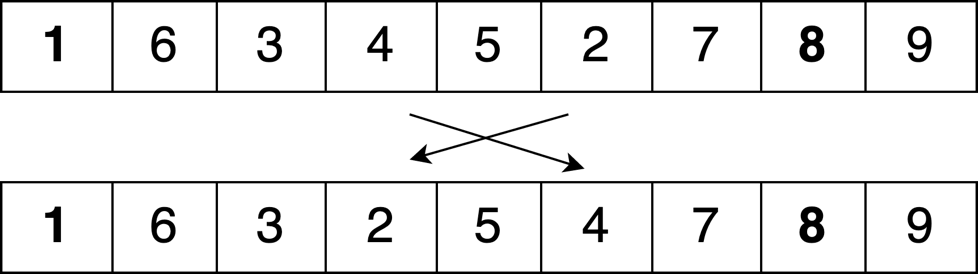


Рисунок 4 – Пример обмена 2-х элементов местами в пределах блока. Жирным обозначены фиксированные элементы.

Если обмен 2-х элементов местами перестал улучшать целевую функцию, то переходим к последнему методу генерации соседей, в котором производится симметричный обмен местами элементов относительно некоторого опорного элемента (рисунок 5).

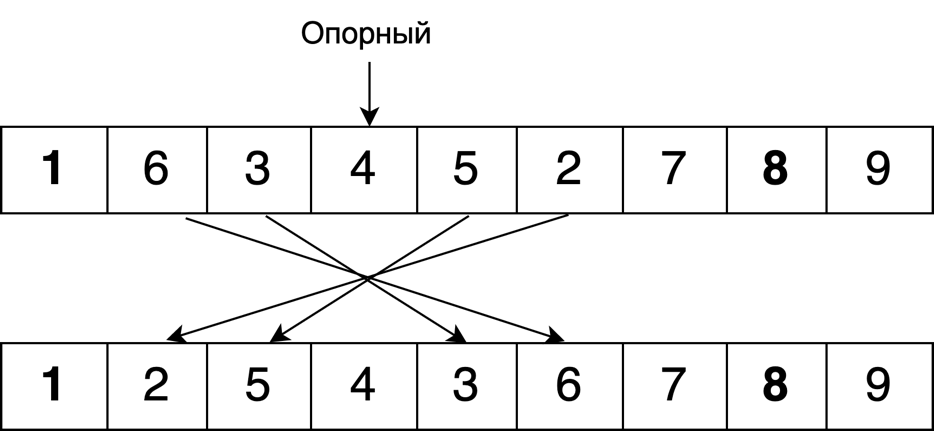


Рисунок 5 – Пример обмена элементов местами относительно опорного в пределах блока. Жирным обозначены фиксированные элементы.

1. В случае, если все методы генерации исчерпаны, и алгоритм не смог выйти из локального минимума, то происходит переход на 1 этап.

**Особенности реализации**

Для написания программной реализации был использован язык программирования Go и интегрированная среда разработки Goland.

Для хранения текущего состояния поля была реализована структура Sudoku, имеющая следующие поля:

* field – массив значений ячеек;
* size – размер поля;
* subSize – размер блоков поля (для судоку 16x16 subSize равно 4).

Поле field хранит значения ячеек в 32-х битном целочисленном формате без знака, представленном на рисунке 6.

Рисунок 6 – Формат хранения значений ячеек поля судоку.

Представленный формат позволяет выполнять проверку удовлетворения ограничений и определять домены переменных с использованием побитовых операций, что ускоряет работу программы и минимизирует использование дополнительной памяти.

Для структуры Sudoku реализованы следующие методы:

* NewSudoku – конструктор структуры, читающий начальное состояние из csv файла;
* Solve – метод, возвращающий решение судоку. Реализует решение с помощью жадного локального поиска с переменным соседством;
* insert, swap, megaswap – методы, возвращающие лучшие соседние состояния, полученные методами циклического сдвига, обмена местами 2-х элементов и обмена местами относительно оперного элемента соответственно.

Результат работы программы

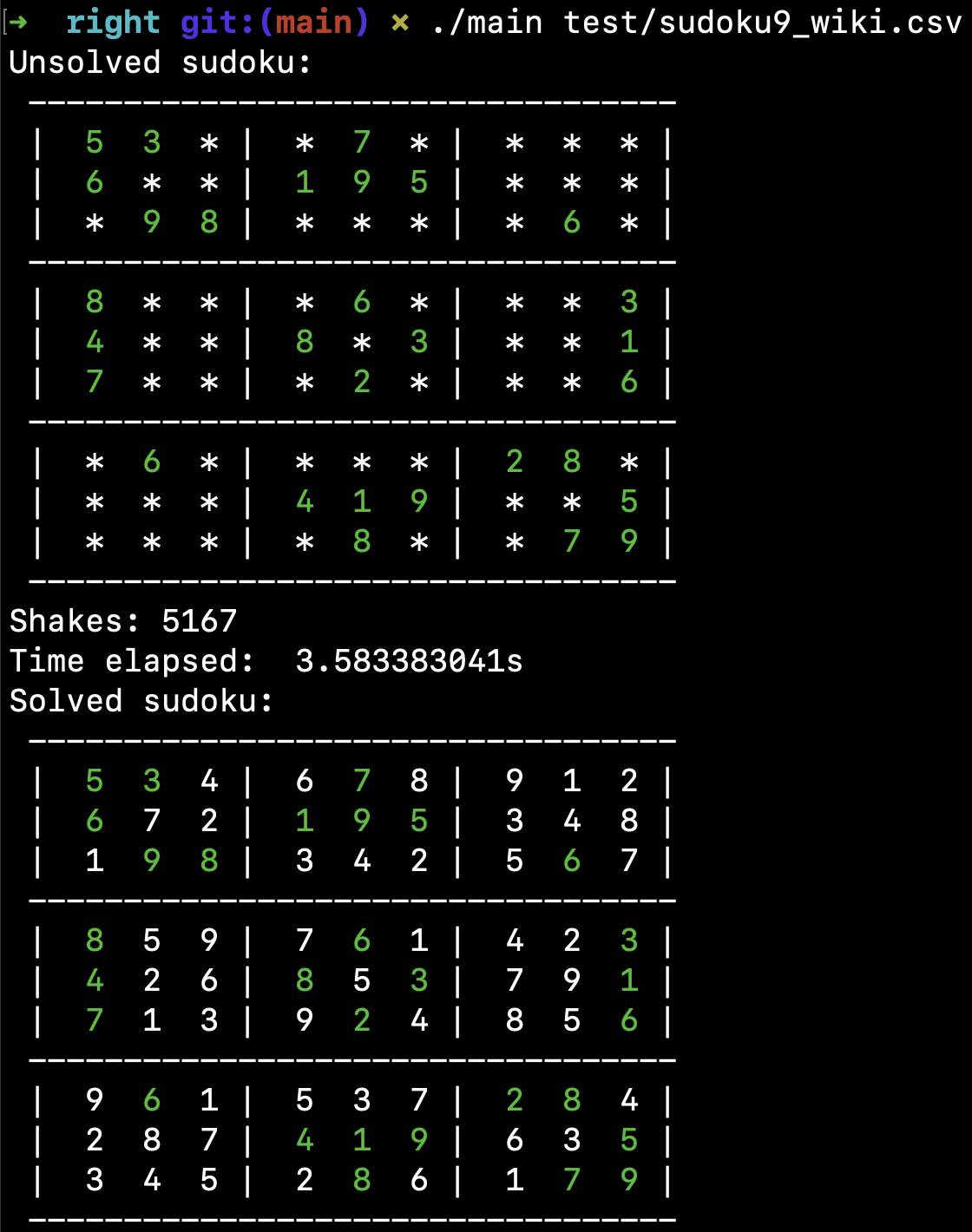


Рисунок 7 – Результат работы программы для судоку 9х9.

Приложение

Листинг 1. Исходный код программы.

**main.go**

**package** main

**import** (

"fmt"

"os"

"right/sudoku"

"time"

)

func main() {

**if** len(os.Args) < 2 {

println("usage: ./main <path\_to\_csv>")

**return**

}

s := sudoku.NewSudoku(os.Args[1])

fmt.Print("Unsolved sudoku:**\n**")

s.PrintSudoku(**true**)

start := time.Now()

solution := s.Solve()

finish := time.Since(start)

fmt.Println("Time elapsed: ", finish)

fmt.Print("Solved sudoku:**\n**")

solution.PrintSudoku(**false**)

}

**sudoku/sudoku.go**

**package** sudoku

**import** (

"encoding/csv"

"fmt"

"math"

"math/rand"

"os"

"strconv"

)

**const** SEED = 123

**type** Sudoku struct {

size int

subSize int

field []uint32

}

func NewSudoku(path string) \*Sudoku {

csvConf, err := os.Open(path)

**if** err != **nil** {

panic(err)

}

**defer** func() {

**if** err = csvConf.Close(); err != **nil** {

panic(err)

}

}()

reader := csv.NewReader(csvConf)

reader.Comma = ' '

reader.TrimLeadingSpace = **true**

reader.FieldsPerRecord = -1

data, err := reader.ReadAll()

**if** err != **nil** {

panic(err)

}

size, \_ := strconv.Atoi(data[0][0])

sudoku := &Sudoku{size: size, subSize: int(math.Sqrt(float64(size))), field: make([]uint32, size\*size)}

data = data[1:]

**var** isFilled bool

**for** i, row := **range** data {

**for** j, el := **range** row {

val, \_ := strconv.Atoi(el)

**if** val != 0 {

isFilled = **true**

} **else** {

isFilled = **false**

}

sudoku.field[i\*size+j] = getBinaryFromInt(val, isFilled, size)

}

}

**return** sudoku

}

func (s \*Sudoku) Solve() \*Sudoku {

r := rand.New(rand.NewSource(SEED))

*// Fill sub-grids*

s.initField()

h := s.heuristic()

neighbourBlocks := 1

currMethod := 0

methods := []func(int, int, int) \*Sudoku{s.insert, s.swap, s.megaswap}

shackesNum := 0

**for** h != 0 {

fmt.Printf("Heuristic: %3d Blocks: %d", h, neighbourBlocks)

**var** best \*Sudoku

oldH := h

**for** i := 0; i < neighbourBlocks; i++ {

s1 := methods[currMethod](i%s.subSize, i/s.subSize, h)

**if** h1 := s1.heuristic(); h1 < h {

best = s1

h = h1

}

}

**if** h == oldH {

neighbourBlocks++

**if** neighbourBlocks > s.size {

neighbourBlocks = s.size

currMethod++

**if** currMethod == len(methods) {

currMethod = 0

neighbourBlocks = 1

s.shake(r)

shackesNum++

}

}

} **else** {

s.field = best.field

}

h = s.heuristic()

fmt.Print("\033[1K**\r**")

}

fmt.Printf("Shakes: %d**\n**", shackesNum)

**return** s

}

func (s \*Sudoku) shake(r \*rand.Rand) {

**for** i := 0; i < s.subSize; i++ {

**for** j := 0; j < s.subSize; j++ {

alreadyInserted := make(map[int]struct{})

**for** k := 0; k < s.subSize; k++ {

**for** l := 0; l < s.subSize; l++ {

**if** isStatic(s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l], s.size) {

alreadyInserted[getIntFromBinary(s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l], s.size)] = struct{}{}

}

}

}

*//count := 1*

**for** k := 0; k < s.subSize; k++ {

**for** l := 0; l < s.subSize; l++ {

**if** isStatic(s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l], s.size) {

**continue**

}

**for** {

v := r.Int()%s.size + 1

\_, exists := alreadyInserted[v]

**if** !exists {

s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l] = getBinaryFromInt(v, **false**, s.size)

alreadyInserted[v] = struct{}{}

**break**

}

}

}

}

}

}

}

*// Invert*

func (s \*Sudoku) invert(i, j int) \*Sudoku {

*// Get copy of field*

res := &Sudoku{size: s.size, subSize: s.subSize, field: make([]uint32, 0)}

res.field = append(res.field, s.field**...**)

*// Get indexes of non-fixed elements*

a := make([]struct {

k int

l int

}, 0)

**for** k := 0; k < s.subSize; k++ {

**for** l := 0; l < s.subSize; l++ {

**if** !isStatic(s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l], s.size) {

a = append(a, struct {

k int

l int

}{k: k, l: l})

}

}

}

start := 0

finish := len(a) - 1

**for** start < finish {

sk, sl := a[start].k, a[start].l

fk, fl := a[finish].k, a[finish].l

tmp := res.field[i\*s.subSize\*s.size+sk\*s.size+j\*s.subSize+sl]

res.field[i\*s.subSize\*s.size+sk\*s.size+j\*s.subSize+sl] = res.field[i\*s.subSize\*s.size+fk\*s.size+j\*s.subSize+fl]

res.field[i\*s.subSize\*s.size+fk\*s.size+j\*s.subSize+fl] = tmp

start++

finish--

}

**return** res

}

func (s \*Sudoku) insert(i, j int, target int) \*Sudoku {

a := make([]struct {

k int

l int

}, 0)

**for** k := 0; k < s.subSize; k++ {

**for** l := 0; l < s.subSize; l++ {

**if** !isStatic(s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l], s.size) {

a = append(a, struct {

k int

l int

}{k: k, l: l})

}

}

}

res := &Sudoku{size: s.size, subSize: s.subSize}

res.field = append(res.field, s.field**...**)

tmp := &Sudoku{size: s.size, subSize: s.subSize}

**for** m := 0; m < len(a)-1; m++ {

**for** n := m + 1; n < len(a); n++ {

tmp.field = make([]uint32, 0)

tmp.field = append(tmp.field, res.field**...**)

start, finish := m, n

**for** start < finish {

sk, sl := a[start].k, a[start].l

fk, fl := a[finish].k, a[finish].l

t := tmp.field[i\*s.subSize\*s.size+sk\*s.size+j\*s.subSize+sl]

tmp.field[i\*s.subSize\*s.size+sk\*s.size+j\*s.subSize+sl] =

tmp.field[i\*s.subSize\*s.size+fk\*s.size+j\*s.subSize+fl]

tmp.field[i\*s.subSize\*s.size+fk\*s.size+j\*s.subSize+fl] = t

start++

finish--

}

**if** tmp.heuristic() < target {

res.field = tmp.field

}

}

}

**return** res

}

func (s \*Sudoku) swap(i, j int, target int) \*Sudoku {

a := make([]struct {

k int

l int

}, 0)

**for** k := 0; k < s.subSize; k++ {

**for** l := 0; l < s.subSize; l++ {

**if** !isStatic(s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l], s.size) {

a = append(a, struct {

k int

l int

}{k: k, l: l})

}

}

}

*// Create copy of field*

res := &Sudoku{size: s.size, subSize: s.subSize}

res.field = append(res.field, s.field**...**)

tmp := &Sudoku{size: s.size, subSize: s.subSize}

**for** m := 0; m < len(a)-1; m++ {

**for** n := m + 1; n < len(a); n++ {

tmp.field = make([]uint32, 0)

tmp.field = append(tmp.field, s.field**...**)

start, finish := m, n

sk, sl := a[start].k, a[start].l

fk, fl := a[finish].k, a[finish].l

t := tmp.field[i\*s.subSize\*s.size+sk\*s.size+j\*s.subSize+sl]

tmp.field[i\*s.subSize\*s.size+sk\*s.size+j\*s.subSize+sl] =

tmp.field[i\*s.subSize\*s.size+fk\*s.size+j\*s.subSize+fl]

tmp.field[i\*s.subSize\*s.size+fk\*s.size+j\*s.subSize+fl] = t

**if** tmp.heuristic() < target {

res.field = tmp.field

}

}

}

**return** res

}

func (s \*Sudoku) megaswap(i, j int, target int) \*Sudoku {

a := make([]struct {

k int

l int

}, 0)

**for** k := 0; k < s.subSize; k++ {

**for** l := 0; l < s.subSize; l++ {

**if** !isStatic(s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l], s.size) {

a = append(a, struct {

k int

l int

}{k: k, l: l})

}

}

}

res := &Sudoku{size: s.size, subSize: s.subSize}

res.field = append(res.field, s.field**...**)

tmp := &Sudoku{size: s.size, subSize: s.subSize}

**for** m := 1; m < len(a)-1; m++ {

tmp.field = make([]uint32, 0)

tmp.field = append(tmp.field, s.field**...**)

start, finish := m-1, m+1

**for** start >= 0 && finish <= len(a)-1 {

sk, sl := a[start].k, a[start].l

fk, fl := a[finish].k, a[finish].l

t := tmp.field[i\*s.subSize\*s.size+sk\*s.size+j\*s.subSize+sl]

tmp.field[i\*s.subSize\*s.size+sk\*s.size+j\*s.subSize+sl] =

tmp.field[i\*s.subSize\*s.size+fk\*s.size+j\*s.subSize+fl]

tmp.field[i\*s.subSize\*s.size+fk\*s.size+j\*s.subSize+fl] = t

start--

finish++

}

**if** tmp.heuristic() < target {

res.field = tmp.field

}

}

**return** res

}

*// Fill empty spaces to satisfy sub-grid constraint*

func (s \*Sudoku) initField() {

**for** i := 0; i < s.subSize; i++ {

**for** j := 0; j < s.subSize; j++ {

alreadyInserted := make(map[int]struct{})

**for** k := 0; k < s.subSize; k++ {

**for** l := 0; l < s.subSize; l++ {

v := getIntFromBinary(s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l], s.size)

**if** v != 0 {

alreadyInserted[v] = struct{}{}

}

}

}

count := 1

**for** k := 0; k < s.subSize; k++ {

**for** l := 0; l < s.subSize; l++ {

v := getIntFromBinary(s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l], s.size)

**if** v != 0 {

**continue**

}

**for** {

\_, exists := alreadyInserted[count]

**if** !exists {

s.field[i\*s.subSize\*s.size+k\*s.size+j\*s.subSize+l] = getBinaryFromInt(count, **false**, s.size)

alreadyInserted[count] = struct{}{}

**break**

}

count++

}

}

}

}

}

}

func (s \*Sudoku) heuristic() int {

**var** res int

**var** mask uint32

**for** i := 0; i < s.size; i++ {

mask <<= 1

mask++

}

*// horizontal*

**for** i := 0; i < s.size; i++ {

**var** heuristic uint32

**for** j := 0; j < s.size; j++ {

heuristic |= s.field[i\*s.size+j]

}

res += countZeros(heuristic, mask)

}

*// vertical*

**for** j := 0; j < s.size; j++ {

**var** heuristic uint32

**for** i := 0; i < s.size; i++ {

heuristic |= s.field[i\*s.size+j]

}

res += countZeros(heuristic, mask)

}

**return** res

}

func (s \*Sudoku) PrintSudoku(isUnsolved bool) {

**for** i := 0; i < s.size; i++ {

**if** i%s.subSize == 0 {

fmt.Print(" ")

**for** k := 0; k < (s.size+1)\*3+s.subSize+1; k++ {

fmt.Printf("-")

}

fmt.Println()

}

**for** j := 0; j < s.size; j++ {

**if** j%s.subSize == 0 {

fmt.Print(" |")

}

n := getIntFromBinary(s.field[i\*s.size+j], s.size)

**if** isStatic(s.field[i\*s.size+j], s.size) {

fmt.Print("\033[32m") *// green*

}

**if** isUnsolved {

**if** n == 0 {

fmt.Printf(" \*")

} **else** {

fmt.Printf(" %2d", n)

}

} **else** {

fmt.Printf(" %2d", n)

}

**if** isStatic(s.field[i\*s.size+j], s.size) {

fmt.Print("\033[0m") *// green*

}

}

fmt.Print(" |")

fmt.Print("**\n**")

}

fmt.Print(" ")

**for** k := 0; k < (s.size+1)\*3+s.subSize+1; k++ {

fmt.Printf("-")

}

fmt.Println()

}

*// Functions to work with binary*

func getIntFromBinary(b uint32, max int) int {

b &= ^(1 << max)

**var** res int

**for** b != 0 {

res++

b >>= 1

}

**return** res

}

func getBinaryFromInt(n int, isStatic bool, max int) uint32 {

**if** n == 0 {

**return** 0

}

**var** res uint32 = 1

**for** i := 0; i < n-1; i++ {

res <<= 1

}

**if** isStatic {

res |= 1 << max

}

**return** res

}

func isStatic(b uint32, max int) bool {

**return** b&(1<<max) != 0

}

func countZeros(b uint32, mask uint32) int {

**var** res int

b &= mask

**for** b != 0 {

**if** b&1 != 1 {

res++

}

b >>= 1

}

**return** res

}