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	CSP

Студент _____ **Макаров Т.Г.**
подпись, дата *фамилия, и.о.*

Преподаватель _____ **Божко А.Н.**
подпись, дата *фамилия, и.о.*

Оценка _____

Москва, 2023 г.

Оглавление

Описание задачи

Вариант 5

9	4		6	1			5	13		15	8			2	
			10			2	7		1		3		11	9	
	3				10				14	5				16	6
13		16	15		3						10	8		12	
				10	13	14					1	3	2		
	16	14								6		7			
12	13		4			8	2	16	3			5			11
			5		1					7	15			10	12
2		9					14	6				16			
		13			12					2			14		10
		5			16					9				3	1
	7	15						12	11						
10						3				4		2			
	6					16		5	2			9			
			12	5	14			15						6	13
7		1		15				14	12						

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

Алгоритм: CSP с использованием бэктрекинга.

Constraint Satisfaction Problem

Constraint Satisfaction Problem (CSP) - задача удовлетворения ограничений. Формально определяется, как тройка <X, D, C>:

- X – множество переменных;
- D – множество доменов, соответствующих элементам X;
- C – множество ограничений.

Каждая из переменных из Xi может принимать значения из Di.

Одним из самых распространённых методов решения задач удовлетворения ограничений является бэктрэкинг.

Бэктрэкинг – рекурсивный алгоритм. Изначально переменным не присвоены какие-либо значения. На каждом шаге выбирается одна переменная, ей последовательно присваиваются значения из домена этой

переменной. Для каждого присвоенного значения проводится проверка удовлетворения ограничений. В случае успеха производится рекурсивный вызов. В случае, если невозможно выполнить присваивание, выполняется откат на шаг назад.

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 1.

$$0 = 0000 \dots 0000\ 0000\ 0000\ 0000$$
$$1 = 0000 \dots 0000\ 0000\ 0000\ 0001$$
$$2 = 0000 \dots 0000\ 0000\ 0000\ 0010$$
$$\dots$$
$$15 = 0000 \dots 0100\ 0000\ 0000\ 0000$$
$$16 = 0000 \dots 1000\ 0000\ 0000\ 0000$$

Рисунок 1 – 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 как CSP с использованием бэктрэкинга;
- getNeighbours – возвращает массив состояний поля судоку, полученный инициализацией ячейки с самым маленьким доменом;
- forwardCheck – инициализирует ячейки, домен которых имеет только одно значение, пока таких ячеек не останется.

Результат работы программы

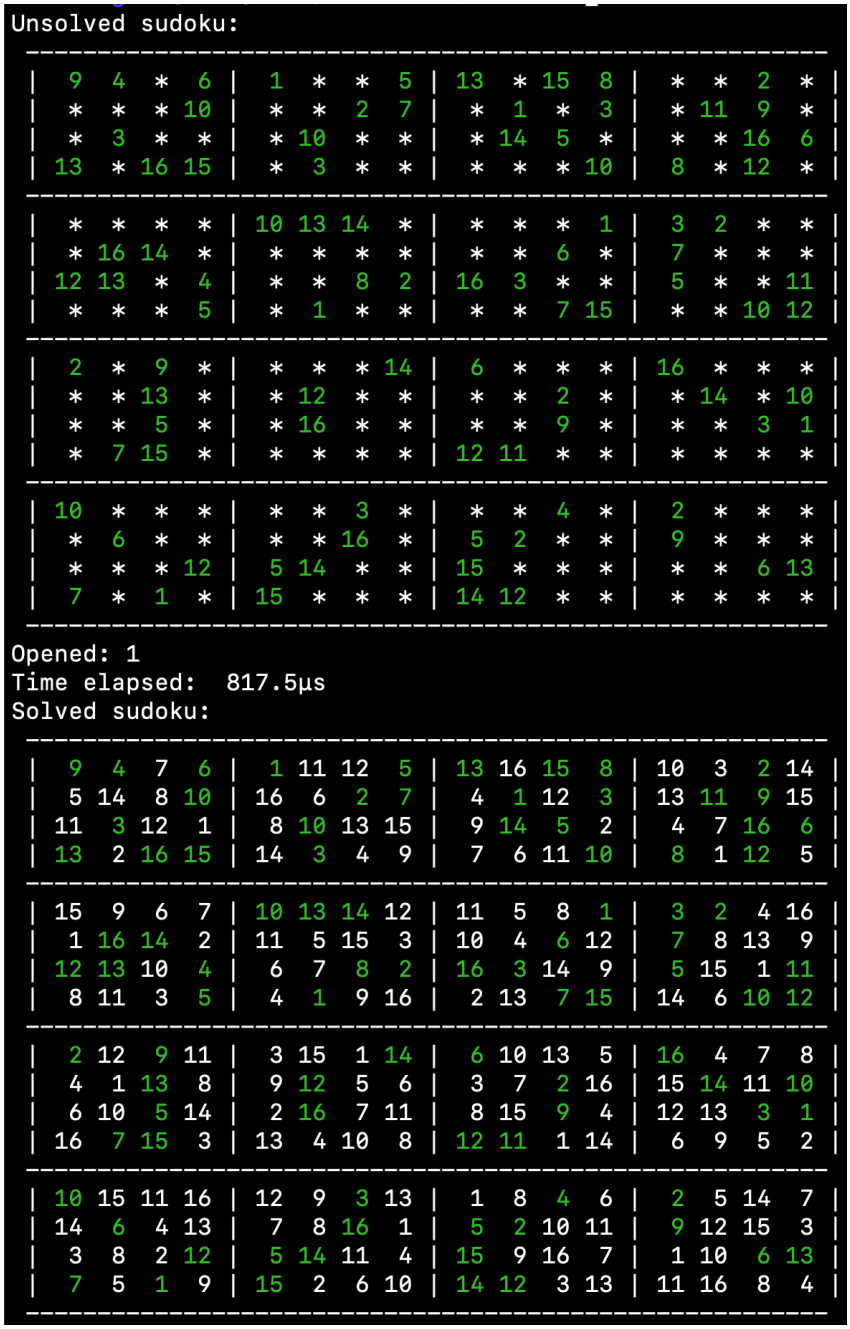

Рисунок 2 – Результат работы программы для судоку в соответствии с вариантом.

Приложение

Листинг 1. Исходный код программы.

main.go

```
package main

import (
        "fmt"
        "os"
        "sudoku/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)
        if solution != nil {
                fmt.Print("Solved sudoku:\n")
                solution.PrintSudoku(false)
        } else {
                fmt.Println("Can't solve")
        }
}
```

sudoku/sudoku.go

```
package sudoku

import (
        "encoding/csv"
        "fmt"
        "math"
        "os"
        "strconv"
)

var startFields = make(map[int]struct{})

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:]
        for i, row := range data {
                for j, el := range row {
                        val, _ := strconv.Atoi(el)
                        sudoku.field[i*size+j] = getBinaryFromInt(val, size)
                        if val != 0 {
                                startFields[i*size+j] = struct{}{}
                        }
                }
        }

        return sudoku
}

func (s *Sudoku) Solve() *Sudoku {
        var stack []*Sudoku
        var count int

        stack = append(stack, s)

        for len(stack) != 0 {
                fmt.Print("Opened: ", count)
                curr := stack[len(stack)-1]
                stack = stack[:len(stack)-1]
                count++

                if curr.heuristic() == 0 {
                        fmt.Println()
                        return curr
                }

                neighbours := curr.getNeighbours()
                stack = append(stack, neighbours...)
                fmt.Print("\033[1K\r")
        }

        fmt.Println()

        return nil
}

func (s *Sudoku) getNeighbours() []*Sudoku {
        var neighbourhood []*Sudoku
```

```
        // Get undefined variable with the smallest domain
        var idx int
        var smallestDomain []uint32
        smallestDomainLen := math.MaxInt
        for i := 0; i < len(s.field); i++ {
                if v := getIntFromBinary(s.field[i], s.size); v == 0 {
                        d := extractDomain(

        s.horizontalConstraint(i)|s.verticalConstraint(i)|s.blockConstraint(i)
,
                                s.size)
                        if len(d) < smallestDomainLen {
                                smallestDomain = d
                                smallestDomainLen = len(smallestDomain)
                                idx = i
                        }
                }
        }
        domain := smallestDomain

        // Generate neighbours with forward checking
        for i := 0; i < len(domain); i++ {
                neighbour := &Sudoku{
                        size:    s.size,
                        subSize: s.subSize,
                }
                neighbour.field = append(neighbour.field, s.field...)
                neighbour.field[idx] = domain[i]

                neighbour.forwardCheck()

                neighbourhood = append(neighbourhood, neighbour)
        }

        /*sort.Slice(neighbourhood, func(i, j int) bool {
                return neighbourhood[i].heuristic() <
neighbourhood[j].heuristic()
        })*/

        return neighbourhood
}

func (s *Sudoku) forwardCheck() {
        for i := 0; i < len(s.field); i++ {
                if v := getIntFromBinary(s.field[i], s.size); v == 0 {
                        domain := extractDomain(

        s.horizontalConstraint(i)|s.verticalConstraint(i)|s.blockConstraint(i)
,
                                s.size)
                        if len(domain) == 1 {
                                s.field[i] = domain[0]
                                i = 0
                        }
                }
        }
}
```

sudoku/constraints.go

```
package sudoku
```

```
func (s *Sudoku) verticalConstraint(idx int) uint32 {
        var res uint32

        j := idx % s.size

        for k := 0; k < s.size; k++ {
                res |= s.field[s.size*k+j]
        }

        return res
}

func (s *Sudoku) horizontalConstraint(idx int) uint32 {
        var res uint32

        i := idx / s.size

        for k := 0; k < s.size; k++ {
                res |= s.field[s.size*i+k]
        }

        return res
}

func (s *Sudoku) blockConstraint(idx int) uint32 {
        var res uint32

        // Calculate block indexes
        i := idx / (s.subSize * s.size)
        j := (idx % s.size) / s.subSize

        for k := 0; k < s.subSize; k++ {
                for l := 0; l < s.subSize; l++ {
                        res |=
s.field[i*s.subSize*s.size+k*s.size+j*s.subSize+l]
                }
        }

        return res
}

func (s *Sudoku) heuristic() int {
        var res int
        // 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, s.size)
        }

        // 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, s.size)
        }

        // block
        for i := 0; i < s.subSize; i++ {
```

```
                for j := 0; j < s.subSize; j++ {
                        var heuristic uint32
                        for k := 0; k < s.subSize; k++ {
                                for l := 0; l < s.subSize; l++ {
                                        heuristic |=
s.field[i*s.subSize*s.size+k*s.size+j*s.subSize+l]
                                }
                        }
                        res += countZeros(heuristic, s.size)
                }
        }

        return res
}
```

sudoku/utils.go

```
package sudoku

import "fmt"

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 := startFields[i*s.size+j]; isStatic {
                                fmt.Print("\033[32m") // green
                        }
                        if isUnsolved {
                                if n == 0 {
                                        fmt.Printf("  *")
                                } else {
                                        fmt.Printf(" %2d", n)
                                }
                        } else {
                                fmt.Printf(" %2d", n)
                        }
                        if _, isStatic := startFields[i*s.size+j]; isStatic {
                                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()
}

func extractDomain(bin uint32, max int) []uint32 {
        var res []uint32
```

```
        for i := 0; i < max; i++ {
                isFilled := bin & 1
                if isFilled == 0 {
                        res = append(res, getBinaryFromInt(i+1, max))
                }
                bin >>= 1
        }

        return res
}

// Functions to work with binary
func getIntFromBinary(b uint32, max int) int {
        var res int
        for b != 0 {
                res++
                b >>= 1
        }

        return res
}

func getBinaryFromInt(n int, max int) uint32 {
        if n == 0 {
                return 0
        }

        var res uint32 = 1
        for i := 0; i < n-1; i++ {
                res <<= 1
        }

        return res
}

func countZeros(b uint32, max int) int {
        var res int

        for i := 0; i < max; i++ {
                if b&1 != 1 {
                        res++
                }
                b >>= 1
        }

        return res
}
```