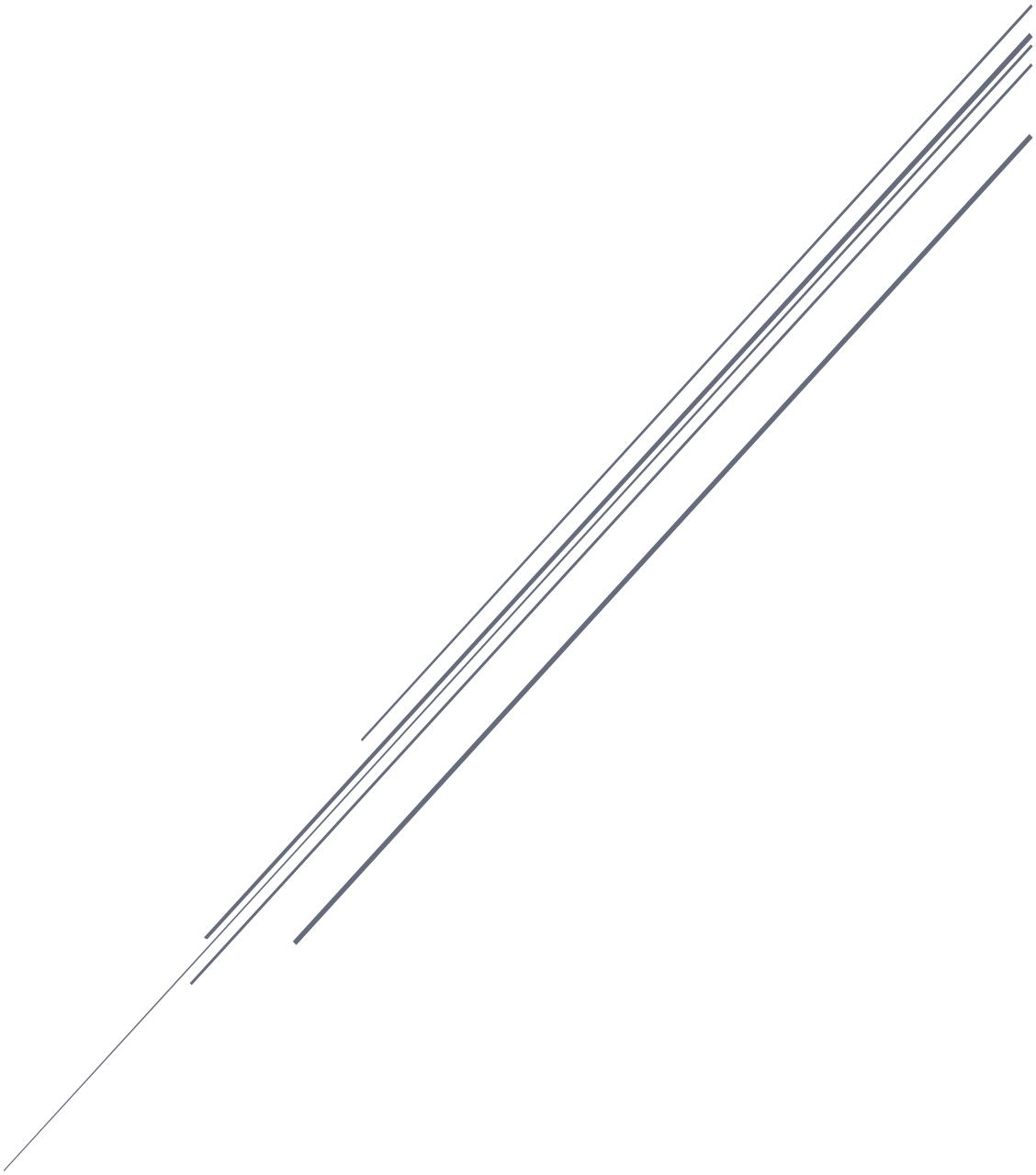
SUDOKU

Relazione del progetto di Esperienze di Programmazione



Lorenzo Arcidiacono
Matr. 534235

SOMMARIO

Descrizione del Problema **3**

Algoritmi di Risoluzione **4**

Backtracking 4

Dancing Links 4

Scelta del linguaggio di programmazione **5**

Scelte implementative **6**

Test **7**

Risultati dei test 7

Bibliografia **8**

Appendice **9**

DESCRIZIONE DEL PROBLEMA

Il sudoku è un gioco di logica che si presenta come una griglia di nove righe orizzontali e nove colonne verticali suddivise in nove sotto-griglie, chiamate regioni, costituite da 3×3 celle. Scopo del gioco è quello di riempire le caselle vuote con numeri da 1 a 9, in modo tale che in ogni riga, colonna e sotto-griglia siano presenti tutte le cifre da 1 a 9 senza ripetizioni. Questo rompicapo, per quanto semplice possa apparire rappresenta un problema matematico di grande complessità.
Il numero delle soluzioni del Sudoku classico è 6.670.903.752.021.072.936.960, approssimativamente $6{,}67 \cdot 10^{21}$. Il numero delle soluzioni sostanzialmente diverse escludendo le simmetrie dovute a rotazioni, riflessioni e permutazioni è 5.472.730.538. (Wikipedia, s.d.)

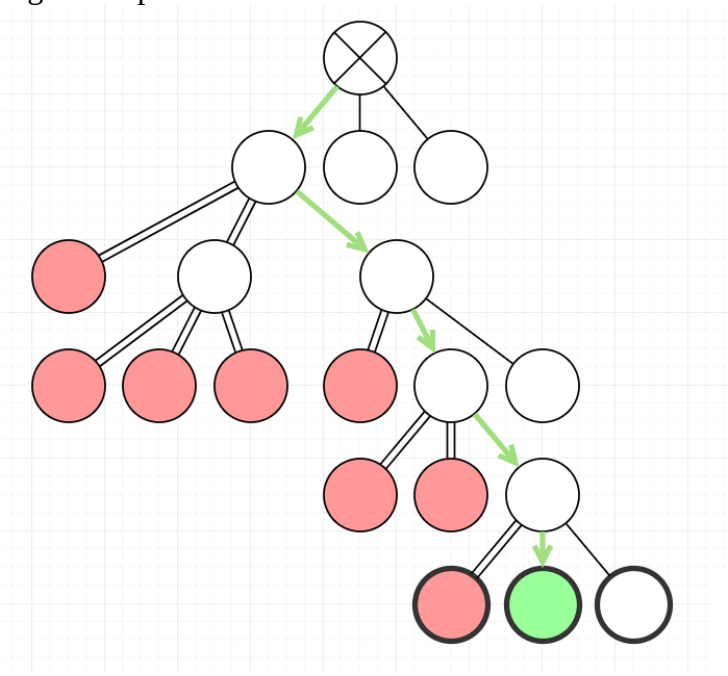
In questo lavoro si cerca di creare un programma che permetta sia di generare una griglia di gioco e poter cercare di completarla, sia di trovare automaticamente una soluzione ad uno schema presentato.
Per quanto riguarda la ricerca della soluzione sono stati presi in considerazione due algoritmi e ne è stato fatto un confronto a livello di tempo di elaborazione.

Algoritmi di Risoluzione

Tra i possibili metodi di risoluzione del sudoku sono stati presi in considerazione in particolare due algoritmi: quello di backtracking e quello denominato Dancing Links.

Backtracking

L'algoritmo di backtracking assegna un valore a una casella dopo aver verificato le condizioni e a partire da questo, ricorsivamente, verifica se si arriva a una soluzione. In caso contrario cancella il valore inserito e ne prova un altro. Se nessun valore porta a una soluzione restituisce un errore. Crea quindi un albero dove la radice è la griglia di partenza, ogni nodo è una possibile scelta e ogni foglia una possibile soluzione.



Esempio di un possibile albero di decisione di un sudoku.

Dancing Links

Una griglia di un sudoku può essere vista come un problema della copertura esatta che in questo caso può essere espresso così: data una matrice di 1 e 0, si scelga una sottomatrice S per le righe così che ogni colonna abbia esattamente un solo 1 per ogni riga di S.
L'algoritmo Dancing Links è stato ideato da Knuth per risolvere questi problemi riducendo drasticamente i tempi di elaborazione. (Knuth)

Un'analisi dell'algoritmo Dancing Links applicata al sudoku è stata svolta come tesi di laurea da Mattias Harrysson e Hjalmar Laestander. (Harrysson & Laestander, 2014)

SCELTA DEL LINGUAGGIO DI PROGRAMMAZIONE

Il linguaggio di programmazione scelto per questo progetto è il linguaggio open source di Google, *Go*, che è stato distribuito nel 2009.
Secondo gli sviluppatori, l'esigenza di creare un nuovo linguaggio di programmazione nasce dal fatto che ai tempi non esisteva un linguaggio di programmazione che soddisfacesse le esigenze di una compilazione efficiente, di una esecuzione veloce e di una facilità di programmazione.
I programmatori che potevano, sceglievano la facilità rispetto alla sicurezza e all'efficienza passando a linguaggi tipicamente dinamici come Python e JavaScript piuttosto che C++ o, in misura minore, Java. (Golang, s.d.)
Il linguaggio di programmazione Go è un progetto open source di Google che vuole essere espressivo, conciso, pulito ed efficiente.
Alcune sue caratteristiche principali sono:

- La concorrenza è applicata al livello nativo tramite le "gorutine", semplici funzioni eseguite concorrentemenete.
- Offre un'elevata velocità di compilazione e di esecuzione.
- Presenta un garbage collector efficiente.
- Il tipaggio statico insieme con un maggior controllo della memoria permette una maggiore sicurezza.
- Non è propriamente un linguaggio ad oggetti, anche se Go ha tipi e metodi e consente uno stile di programmazione orientato agli oggetti, non esiste una gerarchia dei tipi. Ci sono anche modi per incorporare tipi in altri tipi per fornire qualcosa di analogo, ma non identico, alle sottoclassi.

La mascotte di Go

SCELTE IMPLEMENTATIVE

Il codice è stato suddiviso in diversi package per separare le diverse funzionalità:

- Analyzetime: contiene le funzioni per la stampa dei tempi di elaborazione.
- Main: permette di selezionare se lanciare il gioco normale o i casi di test.
- Parse: gestisce la lettura delle griglie dai file.
- Setting: racchiude alcune variabili che possono essere lette da tutti gli altri package e servono a settare i file di input ed output dei test.
- Sudoku: ragguppa sia gli algoritmi per creare e risolvere una griglia, sia un sotto package con gli eventuali errori che possono verificarsi.
- Terminal: raccoglie le funzioni per una gestione più pulita delle stampe a schermo.
- Test: racchiude le funzioni per il testing.

La griglia è stata pensata come una matrice di elementi di tipo cell: ogni elemento ha due campi, un int8, è stato scelto un intero che occupasse minor memoria possibile, in cui viene salvato il valore e un booleano che è true se il valore non è modificabile, cioè è uno degli indizi iniziali.

Per quanto riguarda l'algoritmo Dancing Links è importante sottolineare che: dal momento che l'implementazione risultava particolarmente lunga e complicata è stato scelto di usare quella del vincitore della Go Challenge, una sfida tra programmatori Go, relativa proprio a questo algoritmo. (Rozanski, 2015)

TEST

I test possono essere impostati dal file setting.go, dove possono essere scelti i file di ouput per l'analisi del tempo computazionale e i file di input dei test. Per scegliere di testare questi file bisognerà selezionale l'opzione 'standard' all'avvio del programma.

Ogni file di test proposto presenta un diverso livello di difficoltà, durante il testing viene calcolato il tempo di risoluzione di ogni singola griglia sia con l'algoritmo di Backtracking sia con Dancing Links. Di ogni griglia viene anche calcolato il numero di indizi iniziali.
Si controlla che le soluzioni dei due algoritmi siano uguali fra loro al fine di poter verificare che la griglia sia valida e con un'unica soluzione.
I tempi di elaborazione e il numero di indizi vengono salvati su dei file esterni che verranno sovrascritti all'avvio del test successivo.

Se viene selezionato un unico file di input saranno calcolati direttamente dal programma, e stampati a schermo, la media dei tempi, il tempo più lungo e quello più corto per ogni algoritmo.

I file di test proposti sono composti da 100 griglie ciascuno, il tempo di calcolo complessivo risulta quindi molto lungo e poco pratico per vedere dei risultati velocemente.
È stato quindi aggiunto il file fastTest.txt con poche griglie di varie difficoltà al fine di vedere velocemente lo svolgimento dei test.

Il file hardest.txt presenta alcuni dei sudoku ritenuti più difficili, tra cui i primi due sono considerati in assoluto i sudoku più complicati mai scritti.

I file denominati 'testEasy.txt','testMedium.txt' e 'testRand.txt' con i rispettivi file di soluzione possono essere provati selezionando l'opzione 'nuovo' all'avvio del programma.

RISULTATI DEI TEST

- simple.txt:
  - backtracking: average 1.1640, max value 42.1500,min value 0.0018
  - dancing links: average 0.0185, max value 0.6202,min value 0.0003
- easy.txt:
  - backtracking: average 2.4835, max value 63.0900,min value 0.0068
  - dancing links: average 0.0314, max value 0.7002,min value 0.0004
- intermediate.txt:
  - backtracking: average 2.0046, max value 36.4000,min value 0.0016
  - dancing links: average 0.0222, max value 0.2037,min value 0.0004
- expert.txt:
  - backtracking: average 1.5548, max value 39.3200,min value 0.0037
  - dancing links: average 0.0200, max value 0.4552,min value 0.0003

Come prevedibile i tempi dell'algoritmo Dancing Links sono di molto inferiori rispetto a quelli del Backtracking.
È interessante notare che spesso i test indicati come facili, quindi con un maggior numero di indizi, risultino più lunghi da risolvere che quelli con pochi indizi. Probabilmente quindi per il calcolatore risulta più facile partire da uno schema più "vuoto".

BIBLIOGRAFIA

Golang. (s.d.). *Golang*. Tratto da Go FAQ: https://golang.org/doc/faq#What_is_the_purpose_of_the_project

Harrysson, M., & Laestander, H. (2014). *Solving Sudoku efficiently with Dancing Links*. Stoccolma: KTH Computer Science and Comunication.

Knuth, D. E. (s.d.). Dancing Links. Stanford University.

Rozanski, J. (2015, Novembre 24). *GitHub*. Tratto da Soluzione Go Challenge n.8: https://github.com/golangchallenge/GCSolutions/tree/master/nov15/normal/jakub-rozanski/jrozansk-go-challenge8-d4d18058ef2c

Wikipedia. (s.d.). *Wikipedia*. Tratto da Pagina Wikipedia relativa al sudoku: https://it.wikipedia.org/wiki/Sudoku

Appendice

Codice package main

```
package main
import (
    "fmt"
    "strings"

    "./menu"
    "./test"
)
func main() {
    var sel string
    fmt.Print("Vuoi giocare o eseguire un test? [giocare/test]:")
    fmt.Scanf("%v", &sel)
    sel = strings.ToLower(sel)
    switch sel {
    case "giocare":
        menu.Start()
    case "test":
        test.Start()
    default:
        fmt.Println("Selenzione non valida.")
    }
}
```

Codice package parse

```
package parse
import (
    "bufio"
    "errors"
    "os"
    "strings"
)
//ErrPath errore nel caso il path passato sia una stringa vuota
var ErrPath = errors.New("Il path passato non è valido")

//Parse restituisce un array di stringhe lette dal file per riga
//ha come argomenti il path del file da cui leggere e una stringa in cui scrivere la
stringa con cui iniziano le frasi da saltare
func Parse(path string, delim string) ([]string, error) {
    var red []string
    if path == "" {
        return nil, ErrPath
    }
    file, err := os.Open(path)
    if err != nil {
        return nil, err
    }
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        str := scanner.Text()
        if strings.Index(str, delim) != 0 && str != "" {
            red = append(red, str)
        }
    }
    file.Close()
    return red, nil
}
```

Codice package analyzetime

```go
package analyzetime
import (
    "bufio"
    "fmt"
    "io"
    "os"
    "strconv"
    "time"
)
//Track stampa sul file i secondi passati da start
func Track(start time.Time, str string, f io.Writer) {
    t1 := time.Now()
    fmt.Fprintf(f, "%.4v\n", t1.Sub(start).Seconds())
}
//Average restituisce la media tra i valori letti dal file relativo a path
func Average(path string) (float64, error) {
    file, err := os.Open(path)
    if err != nil {
        return -1, err
    }
    defer file.Close()
    var total, count float64
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        last, err := strconv.ParseFloat(scanner.Text(), 64)
        if err != nil {
            return -1, err
        }
        count++
        total += last
    }
    if err := scanner.Err(); err != nil {
        return -1, err
    }
    return total / count, nil
}
//Max restituisce il valore più grande tra i numeri letti dal file relativo a path
func Max(path string) (float64, error) {
    file, err := os.Open(path)
    if err != nil {
```

```
        return -1, err
    }
    defer file.Close()

    var max, last float64
    scanner := bufio.NewScanner(file)
    max, err = strconv.ParseFloat(scanner.Text(), 64)
    if err != nil {
        return -1, err
    }
    for scanner.Scan() {
        last, err = strconv.ParseFloat(scanner.Text(), 64)
        if err != nil {
            return -1, err
        }
        if last > max {
            max = last
        }
    }
    if err := scanner.Err(); err != nil {
        return -1, err
    }
    return max, nil
}
//Min restituisce il valore più piccolo tra i numeri letti dal file relativo a path
func Min(path string) (float64, error) {
    file, err := os.Open(path)
    if err != nil {
        return -1, err
    }
    defer file.Close()

    var min, last float64
    scanner := bufio.NewScanner(file)
    min, err = strconv.ParseFloat(scanner.Text(), 64)
    if err != nil {
        return -1, err
    }
    for scanner.Scan() {
        last, err = strconv.ParseFloat(scanner.Text(), 64)
```

```
        if err != nil {
            return -1, err
        }
        if min > last {
            min = last
        }
    }
    if err := scanner.Err(); err != nil {
        return -1, err
    }
    return min, nil
}
```

Codice package menu

```
package menu
import (
    "errors"
    "fmt"
    "os"
    "strconv"
    "strings"

    "../parse"
    "../sudoku"
    "./game"
)
var errExit = errors.New("Chiusura del gioco")

//recovery controlla se è stato lanciato un errore e nel caso lo stampa a schermo
func recovery() {
    if e := recover(); e != nil {
        fmt.Println(e)
    }
    os.Exit(1)
}

//getValue legge e restituisce un numero da tastiera
func getValue() (int, error) {
    var i int

    fmt.Print("Enter number: ")
    _, err := fmt.Scanf("%d", &i) //legge un numero da tastiera
    if err != nil {
        return -1, err
    }
    return i, err
}

//getValueString legge una stringa di 9 numeri separati da virgole da tastiera
//restituisce un array dei singoli numeri
func getValueString() ([9]int, error) {
    var val [9]int
    var str string
    _, err := fmt.Scanf("%v", &str) //legge una stringa da tastiera
```

```
    if err != nil {
        return val, err
    }
    single := strings.Split(str, ",") //crea una slice contenente solo i numeri
    for ind := range single {
        val[ind], err = strconv.Atoi(single[ind]) //converti i numeri in int
        if err != nil {
            return val, err
        }
    }
    return val, nil
}

//menu stampa le possibili scelte che l'utente può fare
func menu() (*sudoku.Grid, error) {
    var sel string
    var err error
    var grid *sudoku.Grid
    exit := false
    for exit == false {
        fmt.Println("\n------- MENU -------")
        fmt.Println("Come si desidera generare la griglia? [file/tastiera/random] dgigitare 'exit' per terminare, 'regole' per
leggere le regole del gioco, 'help' per maggiori informazioni")
        fmt.Print("Selezionare l'opzione:")
        fmt.Scanf("%v", &sel)
        sel = strings.ToLower(sel)
        switch sel {
        case "file":
            grid, err = generateFromFile()
            if grid != nil {
                return grid, nil
            }
        case "tastiera":
            grid, err = generateFromInput()
            if grid != nil {
                return grid, nil
            }
        case "random":
            grid, err = generateRandom()
            if grid != nil {
```

```
            return grid, nil
        }
    case "help":
        help()
    case "regole":
        rule()
    case "exit":
        exit = true
    default:
        fmt.Println("Errore nella selezione, scegliere tra 'file/tastiera/random'")
    }
    if err != nil {
        fmt.Println(err)
    }
  }
  return grid, errExit
}

//help stampa a schermo la spiegazione delle varie funzioni
func help() {
  fmt.Println("\n---------- HELP ----------")
  fmt.Println("- Selezionare l'opzione desiderata scrivendo uno tra: file, tastiera, random")
  fmt.Println("- L'opzione file richiede la posizione del file, il carattere usato per le righe di commento, e il carattere usato come casella vuota.")
  fmt.Println("   In base a queste impostazioni legge la griglia dal file (se il file contiene più di una griglia viene letta solo la prima)")
  fmt.Println("- L'opzione tastiera richiede di scrivere manualmente una riga alla volta tutta la griglia.")
  fmt.Println("- L'opzione random richiede di scegliere un livello di difficoltà e genera una griglia in base alla scelta.")
}

//rule stampa a schermo le regole del sudoku
func rule() {
  fmt.Println("------ REGOLE SUDOKU ------")
  fmt.Println("- Scopo del gioco è riempire tutta la griglia.")
  fmt.Println("- Su ogni riga devono esserci tutti i valori da 1 a 9 senza ripetizioni")
  fmt.Println("- Su ogni colonna devono esserci tutti i valori da 1 a 9 senza ripetizioni")
  fmt.Println("- In ogni blocco devono esserci tutti i valori da 1 a 9 senza ripetizioni")
}

//generateFromFile legge una griglia da un file
```

```
func generateFromFile() (*sudoku.Grid, error) {
    var path, delim, void string
    fmt.Print("Indicare il path del file da cui leggere lo schema: ")
    fmt.Scanf("%s", &path)
    fmt.Print("Indicare il carattere che precede le righe da non leggere: ")
    fmt.Scanf("%s", &delim)
    fmt.Print("Indicare il valore che indica il numero mancante (di solito 0): ")
    fmt.Scanf("%s", &void)

    red, err := parse.Parse(path, delim)
    if err != nil {
        return nil, err
    }
    grid, err := sudoku.CreateFromString(red, void)
    if err != nil {
        return nil, err
    }
    return grid, nil
}

//generateFromInput permette all'utente di scrivere la griglia una riga alla volta
func generateFromInput() (*sudoku.Grid, error) {
    var values [9][9]int
    var err error
    fmt.Println("Scrivere una riga alla volta separando i valori con una virgola. Indicare le caselle vuote con 0")
    for i := 0; i < 9; i++ {
        values[i], err = getValueString()
        if err != nil {
            return nil, err
        }
    }
    grid := sudoku.CreateFromArray(values)
    return grid, nil
}

//generateRandom genera una griglia nuova
func generateRandom() (*sudoku.Grid, error) {
    var lev string
    fmt.Println("Selezionare un livello tra: facile, medio, difficile, arduo , impossibile")
    fmt.Scanf("%s", &lev)
```

```
    grid, err := sudoku.Generate(lev)
    if err != nil {
        return nil, err
    }
    return grid, nil
}

//Start avvia il gioco e stampa, all' inizio di ogni sessione di gioco, il menu
func Start() {
    defer recovery() //chiama la funzione recovery al momento dell'uscita dalla funzione Start

    exit := false
    for !exit {
        grid, err := menu()
        if err == errExit { //nel menu viene selezionata l'opzione di uscita
            os.Exit(0)
        }
        if err != nil {
            panic(err)
        }

        game.Game(grid)
    }
}
```

Codice package game

```
package game
import (
    "fmt"
    "os"
    "strconv"
    "strings"

    "../../sudoku"
    "../../terminal"
)

//Game avvia una singola partita
func Game(g *sudoku.Grid) {
    var str string
    exit := false
    rule()
    for !exit {
        g.ShowScheme(os.Stdout)
        fmt.Print("\n»  ")
        fmt.Scanf("%s", &str)
        str = strings.ToLower(str) //si aspetta di leggere una tripla i,j,v dove v->(i,j)
        sub := strings.Split(str, ",")
        _, err := strconv.Atoi(sub[0]) //prova a trasformare i singoli valori in int
        if err == nil {              //se non c'è stato un errore prova a inserire v in (i,j)
            insert(sub, g)
        } else { //altrimenti controlla la stringa
            switch str {
            case "soluzione": //è stata richiesta la soluzione
                solve(g)
                exit = true
            case "exit": //si vuole uscire dal gioco
                terminal.Clear()
                exit = true
            case "regole": //si chiede di stampare a schermo le regole
                terminal.Clear()
                rule()
            default:
                terminal.Clear()
                fmt.Println("Errore nella selezione")
            }
```

```
        }


 if sudoku.IsSolved(*g) && !exit { //prima di ricominciare il ciclo controlla che il sudoku non sia risolto
            fmt.Println("Hai risolto correttamente il sudoku!")
            exit = true
        }
    }
    g.ShowScheme(os.Stdout)
}

//rule stampa a schermo la spiegazione delle possibili scelte
func rule() {
    fmt.Println("------- REGOLE -------")
    fmt.Println("- Scrivendo: soluzione -> viene stampata la soluzione del sudoku.")
    fmt.Println("- Scrivendo: exit -> viene chiuso lo schema attuale.")
    fmt.Println("- Scrivendo: regole -> vengono stampate le regole del gioco.")
    fmt.Println("- Scrivendo una i,j,v -> prova ad inserire v nella cella (i,j).")
    fmt.Println("- La cifra 0 indica la casella vuota.")
    fmt.Println("- Il gioco non permette di scrivere una cifra in una casella se è già presente nel relativo: blocco, riga o
colonna.")
    fmt.Println("- Le caselle vanno da (1,1) a (9,9).")
}

//insert riceve una slice con tre valori numerici che indicano la posizione e il valore da inserire
func insert(str []string, g *sudoku.Grid) {
    r, err := strconv.Atoi(str[0]) //indice di riga
    if err != nil {
        panic(err)
    }
    c, err := strconv.Atoi(str[1]) //indice di colonna
    if err != nil {
        panic(err)
    }
    v, err := strconv.Atoi(str[2]) //valore da inserire
    if err != nil {
        panic(err)
    }
    err = g.Set(r-1, c-1, int8(v)) //vengono passati r-1 e c-1 poichè la griglia è indirizzata a partire da 0
    terminal.Clear()               // mentre i valori passati da 1
```

```
    if err != nil {
        fmt.Println(err)
    }
}

//solve scrive nella griglia passata la soluzione del sudoku
func solve(g *sudoku.Grid) {
    check, _, _ := sudoku.Solve(g)
    if !check {
        fmt.Println("Soluzione non trovata")
    }
}
```

Codice package sudoku

```
package sudoku

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "strings"
    "time"

    "./sdkerror"
)

//Costanti di inizializzazione dello schema.
const (
    rows     = 9
    columns  = 9
    empty    = 0
    maxValue = 9
)

//Tipo che descrive il livello di difficoltà.
type difficulty int

//Enumerazione dei livelli di difficoltà e degli indizi relativi.
const (
    easy       difficulty = 31
    medium     difficulty = 26
    hard       difficulty = 22
    veryHard   difficulty = 20
    impossible difficulty = 17
)

//Cell descrive una singola casella della griglia.
type cell struct {
    val   int8 //numero nella cella; int8 occupa meno memoria.
    fixed bool //false se il valore è modificabile.
}

//Grid descrive una griglia composta da rows*columns cell (esportato).
```

```
type Grid [rows][columns]cell

//setDifficulty restituisce il livello di difficoltà in base alla stringa passata
func setDifficulty(str string) (difficulty, error) {
    str = strings.ToLower(str)
    fmt.Println("letto:", str)
    switch str {
    case "facile":
        return easy, nil
    case "medio":
        return medium, nil
    case "difficile":
        return hard, nil
    case "arduo":
        return veryHard, nil
    case "impossibile":
        return impossible, nil
    default:
        return -1, sdkerror.ErrDifficulty
    }
}

//CreateByEnum crea una griglia a partire dall'enumerazione completa scelta dall'utente (esportata)
func CreateByEnum(digits [rows][columns]int8) *Grid {
    var g Grid
    for i := 0; i < rows; i++ {
        for j := 0; j < columns; j++ {
            d := digits[i][j]
            if d != empty {
                g[i][j].val = d
                g[i][j].fixed = true
            }
        }
    }
    return &g
}

//CreateFromString restituisce una griglia creata a partire da una singola strina (esportata).
func CreateFromString(digits []string, void string) (*Grid, error) {
    var g Grid
```

```
    count := 0
    for index := range digits {
        if count >= rows {
            break
        }
        single := strings.Split(digits[index], "")
        for item := range single {
            if single[item] != void {
                i := index % 9
                value, err := strconv.Atoi(single[item])
                if err != nil {
                    return nil, err
                }
                g[i][item].val = int8(value)
                if value != empty {
                    g[i][item].fixed = true
                }
            }
        }
        count++
    }
    return &g, nil
}

//CreateFromArray restituisce una griglia creata a partire da una matrice (esportata).
func CreateFromArray(digit [9][9]int) *Grid {
    var g Grid
    for i := 0; i < rows; i++ {
        for j := 0; j < columns; j++ {
            if digit[i][j] != empty {
                g[i][j].val = int8(digit[i][j])
                g[i][j].fixed = true
            }
        }
    }
    return &g
}

//Generate restituisce una griglia creata randomicamente (esportata).
func Generate(lev string) (*Grid, error) {
```

```
    level, err := setDifficulty(lev)
    if err != nil {
        return nil, err
    }
    var tab Grid

    SolveBT(&tab)               //risolve una griglia vuota
    randomStart(&tab, int(level)) //imposta alcune caselle a zero in base alla difficoltà scelta
    tab.setUnmodifiable()       //setta le caselle che non possono essere modificate
    return &tab, nil
}

//randomStart elimina dalla griglia i valori di alcune celle in modo da rimanere solo con clues celle impostate.
func randomStart(g *Grid, clues int) {
    rand.Seed(time.Now().UTC().UnixNano())
    for i := rows * columns; i > clues; i-- {
        elim := rand.Intn(maxValue) + 1
        r, c := g.find(int8(elim))
        if r != -1 && c != -1 {
            g.clear(r, c)
        } else {
            i++
        }
    }
}

//find trova il primo valore == digit a partire da una riga generata randomicamente
func (g *Grid) find(digit int8) (int, int) {
    tryAll := false
    count := 0
    rand.Seed(time.Now().UTC().UnixNano())
    i := rand.Intn(rows)
    for !tryAll { //se lo ho già cancellato dalla riga random vado a quella successiva
        for j := 0; j < columns; j++ {
            if g[i][j].val == digit {
                return i, j
            }
        }
        count++
        i = (i + 1) % rows
```

```
        if count == rows-1 { //se digit non appare nella griglia
            tryAll = true
        }
    }
    return -1, -1
}

//setUnmodifiable setta il campo fixed di ogni cella dove val != empty a true.
func (g *Grid) setUnmodifiable() {
    for i := 0; i < rows; i++ {
        for j := 0; j < columns; j++ {
            if g[i][j].val != empty {
                g[i][j].fixed = true
            }
        }
    }
}

//inBound controlla che la coppia row,column appartenga alla griglia.
func inBound(row, column int) bool {
    if row < 0 || row >= rows {
        return false
    }

    if column < 0 || column >= columns {
        return false
    }
    return true
}

//validDigit controlla che il valore passato come argomento sia accettabile
func validDigit(digit int8) bool {
    return digit >= 1 && digit <= 9
}

func (g *Grid) isFixed(row, column int) bool {
    return g[row][column].fixed
}

//inRow restituisce true se nella riga esiste già una cella con val == digit, false altrimenti
```

```
func (g *Grid) inRow(row int, digit int8) bool {
    for c := 0; c < columns; c++ {
        if g[row][c].val == digit {
            return true
        }
    }
    return false
}

//inColumn restituisce true se nella colonna esiste già una cella con val == digit, false altrimenti
func (g *Grid) inColumn(column int, digit int8) bool {
    for r := 0; r < rows; r++ {
        if g[r][column].val == digit {
            return true
        }
    }
    return false
}

//inRegion restituisce true se nella regione esiste già una cella con val == digit, false altrimenti
func (g *Grid) inRegion(row, column int, digit int8) bool {
    startRow, startCol := (row/3)*3, (column/3)*3
    for r := startRow; r < startRow+3; r++ {
        for c := startCol; c < startCol+3; c++ {
            if g[r][c].val == digit {
                return true
            }
        }
    }
    return false
}

//checkAll controlla tutte le condizioni
func (g Grid) checkAll(row, column int, digit int8) error {
    switch {
    case !inBound(row, column):
        return sdkerror.ErrBuonds
    case !validDigit(digit):
        return sdkerror.ErrValue
    case g.isFixed(row, column):
```

```
        return sdkerror.ErrFixed
    case g.inRow(row, digit):
        return sdkerror.ErrRow
    case g.inColumn(column, digit):
        return sdkerror.ErrCol
    case g.inRegion(row, column, digit):
        return sdkerror.ErrRegion
    }
    return nil
}

//Set se le condizioni sono verificate setta la cella (row,column) a digit (esportata).
func (g *Grid) Set(row, column int, digit int8) error {
    err := g.checkAll(row, column, digit)
    if err != nil {
        return err
    }
    g[row][column].val = digit
    return nil
}

//clear setta a empty la cella (row,column).
func (g *Grid) clear(row, column int) error {
    switch {
    case !inBound(row, column):
        return sdkerror.ErrBuonds
    case g.isFixed(row, column):
        return sdkerror.ErrFixed
    }

    g[row][column].val = empty
    return nil
}

//Show stampa una versione semplice della griglia (esportata).
func (g Grid) Show() {
    for i := 0; i < rows; i++ {
        for j := 0; j < columns; j++ {
            fmt.Printf("%v ", g[i][j].val)
        }
```

```
        fmt.Print("\n")
    }
    fmt.Print("\n\n")
}

//ShowScheme stampa una versione elaborata della griglia (esportata).
func (g Grid) ShowScheme(f *os.File) {
    fmt.Print("\n--------SUDOKU------\n\n")
    for i := 0; i < rows; i++ {
        if i == 3 || i == 6 {
            fmt.Fprint(f, "---|---|---\n")
        }
        for j := 0; j < columns; j++ {
            if j == 3 || j == 6 {
                fmt.Fprint(f, "|")
            }
            fmt.Fprint(f, g[i][j].val)
        }
        fmt.Fprint(f, "\n")
    }
}

//toString restituisce la griglia sotto forma di stringa.
func (g Grid) toString() string {
    var str string
    for i := 0; i < rows; i++ {
        for j := 0; j < columns; j++ {
            val := int(g[i][j].val)
            str += strconv.Itoa(val)
        }
    }
    return str
}

//GetVal restituisce il valore della cella (i,j)
func (g Grid) GetVal(i, j int) int {
    return int(g[i][j].val)
}

//IsSolved restituisce true se tutte le celle sono state riempite
```

```
func IsSolved(g Grid) bool {
    i, j := findUnassignedLoc(g)
    if i == -1 && j == -1 {
        return true
    }
    return false
}
```

Codice package sudoku, file solvingAlg.go

```
package sudoku
import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"

    "../analyzetime"
    "../setting"
    "./algorithms/dlx"
)

//Solve cerca la soluzione con entrambi gli algoritmi e ne calcola il tempo di esecuzione
func Solve(g *Grid) (bool, string, string) {
    outputBt, err := os.OpenFile(setting.OutPathBT, os.O_APPEND|os.O_WRONLY|os.O_CREATE, 0666)
    outputDlx, err := os.OpenFile(setting.OutPathDLX, os.O_APPEND|os.O_WRONLY|os.O_CREATE, 0666)

    defer outputBt.Close()
    defer outputDlx.Close()

    if err != nil {
        fmt.Println(err)
        return false, "", ""
    }
    start := time.Now() //setta il tempo di inizio dell'elaborazione
    check, sol1 := SolveDLX(g)
    analyzetime.Track(start, "DLX", outputDlx) //stampa il tempo impiegato
    if check == false {
        return false, "", ""
    }
    start = time.Now()
    check = SolveBT(g)
    analyzetime.Track(start, "BT", outputBt)
    return check, sol1, g.toString()

}

//SolveDLX cerca la soluzione tramite l'algoritmo Dancing Links
func SolveDLX(g *Grid) (bool, string) {
```

```
    var str string
    for i := 0; i < 9; i++ {
        for j := 0; j < 9; j++ {
            val := int(g[i][j].val)
            str += strconv.Itoa(val)
        }
    }
    solver, err := dlx.InitSolver(str)
    if err != nil || !solver.Solve() {
        fmt.Println("Couldn't solve given sudoku: ", err)
        return false, ""
    }
    return true, solver.GetSolution()
}

//SolveBT cerca la soluzione tramite l'algoritmo di Backtracking
func SolveBT(g *Grid) bool {
    i, j := findUnassignedLoc(*g) //cerca una locazione libera
    if i == -1 && j == -1 {
        return true
    }

    values := randValues()
    for index := range values {
        err := g.checkAll(i, j, values[index])
        if err == nil {
            g[i][j].val = values[index]
            if SolveBT(g) { //cerca di risolverlo tramite ricorsione
                return true
            }
            g[i][j].val = empty //se non ci riesce cambia il valore
        }
    }
    return false
}

//randValues restituisce un array con una permutazione di tutti i valori da 1 a maxValue
func randValues() [maxValue]int8 {
    var values [maxValue]int8
    rand.Seed(time.Now().UTC().UnixNano())
```

```
    perm := rand.Perm(maxValue)
    for i := range perm {
        perm[i]++
        values[i] = int8(perm[i])
    }
    return values
}

//findUnassignedLoc restituisce una coppia che indica una posizione vuota nella griglia
func findUnassignedLoc(g Grid) (int, int) {
    for i := 0; i < rows; i++ {
        for j := 0; j < columns; j++ {
            if g[i][j].val == empty {
                return i, j
            }
        }
    }
    return -1, -1
}
```

Codice package sdkerror

```
package sdkerror

import "errors"

var (
    ErrBuonds     = errors.New("Fuori dai limiti")
    ErrValue      = errors.New("Valore errato")
    ErrCol        = errors.New("Cifra già presente in questa colonna")
    ErrRow        = errors.New("Cifra già presente in questa riga")
    ErrRegion     = errors.New("Cifra già presente in questo settore")
    ErrFixed      = errors.New("Casella non modificabile")
    ErrDifficulty = errors.New("Livello di difficoltà sbagliato")
)
```

Codice package dlx, file DLX.go

```
package dlx
// Autore: Jakub Rozanski
// Progetto originale: https://github.com/golangchallenge/GCSolutions/tree/master/nov15/normal/jakub-rozanski/jrozansk-go-challenge8-d4d18058ef2c

type Solver struct {
    solution []int
    header   *Node
}
//InitSolver genera la matrice da usare per trovare la soluzione
func InitSolver(input string) (*Solver, error) {
    if len(input) != NumOfCells {
        return nil, InputTooShortError
    }
    solver := Solver{header: generateDlxMatrix()}
    fixedCells := asciiToInts(input)
    if err := solver.addToSolution(fixedCells); err != nil {
        return nil, err
    }
    return &solver, nil
}
//Solve cerca una soluzione ricororsivamente applicando l'algoritmo 'Dancing Links'
func (s *Solver) Solve() bool {
    col := s.header.Right
    if s.header.Right == s.header { //tutte le possibilità già controllate
        return true
    }
    if col.Down == col {
        return false
    }
    cover(col)
    for row := col.Down; row != col; row = row.Down {
        for cell := row.Right; cell != row; cell = cell.Right {
            cover(cell.Head)
        }
        if s.Solve() { //cerca ricorsimante nell nuovo header più a destra
            s.solution = append(s.solution, row.Row) //se trova una soluzione la salva
            return true
        }
        for cell := row.Left; cell != row; cell = cell.Left {
```

```
            uncover(cell.Head) //se non trova una soluzione fa la uncover dell'header delle righe
        }
    }
    uncover(col) //se non trova una soluzione a partire da questa colonna fa la uncover della colonna
    return false
}
//restituisce la soluzione del problema sotto forma di stringa
func (s *Solver) GetSolution() string {
    result := make([]byte, NumOfCells)
    for _, v := range s.solution {
        row, col, val := decodePosibility(v)
        result[row*GridSize+col] = intToAscii(val) + 1
    }
    return string(result)
}

func (s *Solver) addToSolution(fixedCells []int) error {
    for i, val := range fixedCells {
        if val == 0 {
            continue
        }

        row := i / GridSize
        col := i % GridSize

        constraints := constraintPositions(row, col, val-1)
        for _, columnIdx := range constraints {
            col := getColumnHeader(s.header, columnIdx)
            if col == nil {
                return InputMalformedError
            }
            cover(col) //fa la cover di tutta la colonna a partire dal'header
        }
        s.solution = append(s.solution, encodePossibility(row, col, val-1))
    }
    return nil
}

func cover(col *Node) {
    col.Cover() //cover dell'header
```

```
    coverRows(col)
}

func uncover(header *Node) {
    uncoverRows(header)
    uncoverCol(header)
}

func coverRows(header *Node) {
    for ptr := header.Down; ptr != header; ptr = ptr.Down { //cover di tutta la colonna
        for cell := ptr.Right; cell != ptr; cell = cell.Right { //cover della riga associata
            cell.Cover()
        }
    }
}

func uncoverRows(header *Node) {
    for ptr := header.Up; ptr != header; ptr = ptr.Up {
        for cell := ptr.Left; cell != ptr; cell = cell.Left {
            uncoverRow(cell)
        }
    }
}

func uncoverCol(header *Node) {
    header.Left.Right = header
    header.Right.Left = header
}

func uncoverRow(cell *Node) {
    cell.Up.Down = cell
    cell.Down.Up = cell
}
```

Codice package dlx, file dlxerror.go

```
package dlx
// Autore: Jakub Rozanski
```

```
// Progetto originale: https://github.com/golangchallenge/GCSolutions/tree/master/nov15/normal/jakub-rozanski/jrozansk-go-challenge8-d4d18058ef2c

import "errors"

var (
    InputTooShortError  = errors.New("Input is too short!")
    InputMalformedError = errors.New("Input is malformed!")
)
```

Codice package dlx, file matrix.go

```
package dlx

// Autore: Jakub Rozanski
// Progetto originale: https://github.com/golangchallenge/GCSolutions/tree/master/nov15/normal/jakub-rozanski/jrozansk-go-challenge8-d4d18058ef2c

const (
    GridSize       = 9
    RegionSize     = 3
    ConstraintTypes = 4
    NumOfCells     = GridSize * GridSize
    Constraints    = NumOfCells * ConstraintTypes
    Possibilities  = GridSize * GridSize * GridSize
)

//generateDlxMatrix genera la matrice di copertura esatta
func generateDlxMatrix() *Node {
    rows, cols := generateUnassociatedNodes()

    header := headerRow()
    associateVertical(header.Right, cols)
    associateHorizontal(rows)
    return header
}

//associateHorizzonatl associa i nodi della stessa riga
func associateHorizontal(rows [Possibilities][ConstraintTypes]*Node) {
    for _, nodes := range rows {
        n := len(nodes)
        for i, node := range nodes {
            node.Left = nodes[(i-1+n)%n]
            node.Right = nodes[(i+1)%n]
        }
    }
}

//associateVertical associa i nodi della stessa colonna
func associateVertical(headers *Node, cols [Constraints][GridSize]*Node) {
    header := headers
    for _, nodes := range cols {
```

```
        n := len(nodes)
        for i, node := range nodes {
            //We need pointer to col header from every cell (for efficiency purposes)
            node.Head = header
            node.Up = nodes[(i-1+n)%n]
            node.Down = nodes[(i+1)%n]
        }
        attachHeader(header, nodes[0])
        header = header.Right //scorre l'header da sx a dx
    }
}

//attachHeader collega l'header al nodo
func attachHeader(header *Node, to *Node) {
    header.Up = to.Up
    header.Down = to
    to.Up.Down = header
    to.Up = header
}

//generateUnassociatedNodes genera tutti i nodi senza associarli tra loro
func generateUnassociatedNodes() ([Possibilities][ConstraintTypes]*Node, [Constraints][GridSize]*Node) {
    cols := [Constraints][GridSize]*Node{}
    rows := [Possibilities][ConstraintTypes]*Node{}

    rowCnt := [Possibilities]int{}
    colCnt := [Constraints]int{}

    for row := 0; row < GridSize; row++ {
        for col := 0; col < GridSize; col++ {
            for val := 0; val < GridSize; val++ {
                pos := encodePossibility(row, col, val)
                for _, con := range constraintPositions(row, col, val) {
                    n := NodeRegular(pos, con)
                    rows[pos][rowCnt[pos]] = n
                    cols[con][colCnt[con]] = n
                    rowCnt[pos]++
                    colCnt[con]++
                }
            }
```

```
        }
    }
    return rows, cols
}

func constraintPositions(row int, col int, val int) []int {
    box := getBoxNumber(row, col)
    return []int{
        row*GridSize + col,
        NumOfCells + row*GridSize + val,
        NumOfCells*2 + col*GridSize + val,
        NumOfCells*3 + box*GridSize + val}
}

//headerRow crea la riga degli header
func headerRow() *Node {
    root := NodeHeader(-1)
    last := header(root, 0)
    root.Left = last
    last.Right = root
    return root
}

func header(node *Node, n int) *Node {
    if n == Constraints {
        return node
    }
    node.Right = NodeHeader(n)
    node.Right.Left = node
    return header(node.Right, n+1)
}

func encodePossibility(row int, col int, val int) int {
    return NumOfCells*row + col*GridSize + val
}

func decodePosibility(encoded int) (int, int, int) {
    row := encoded / NumOfCells
    col := (encoded % NumOfCells) / GridSize
    val := (encoded % NumOfCells) % GridSize
```

```
    return row, col, val
}

func getBoxNumber(row int, col int) int {
    return (row/RegionSize)*RegionSize + col/RegionSize
}

func getColumnHeader(start *Node, index int) *Node {
    for ptr := start.Right; ptr != start; ptr = ptr.Right {
        if ptr.Col == index {
            return ptr
        }
    }
    return nil
}
```

Codice package dlx, file misc.go

```
package dlx
// Autore: Jakub Rozanski
// Progetto originale: https://github.com/golangchallenge/GCSolutions/tree/master/nov15/normal/jakub-rozanski/jrozansk-go-challenge8-d4d18058ef2c
import "fmt"
func PrintReadableGrid(sol string) {
    for row := 0; row < GridSize; row++ {
        for col := 0; col < GridSize; col++ {
            fmt.Printf("%c ", sol[row*GridSize+col])
        }
        fmt.Println()
    }
}

func asciiToInts(numbers string) []int {
    toReturn := []int{}
    for _, c := range numbers {
        toReturn = append(toReturn, int(c)-int('0'))
    }
    return toReturn
}
```

```
func intToAscii(input int) byte {
    return byte(input) + byte('0')
}
```

Codice package dlx, file node.go

```
package dlx
// Autore: Jakub Rozanski
// Progetto originale: https://github.com/golangchallenge/GCSolutions/tree/master/nov15/normal/jakub-rozanski/jrozansk-go-challenge8-d4d18058ef2c
//Node struttura che descrive un singolo nodo
type Node struct {
    Left  *Node
    Right *Node
    Up    *Node
    Down  *Node

    Head *Node

    Col int
    Row int
}
//nodeInit inizializza un nodo
func nodeInit(row int, col int) *Node {
    n := Node{}
    n.Left = &n
    n.Right = &n
    n.Up = &n
    n.Down = &n
    n.Row = row
    n.Col = col
    return &n
}
//NodeRegular inizializza un nodo qualsiasi
func NodeRegular(row int, col int) *Node {
    return nodeInit(row, col)
}
//NodeHeader inizializza un nodo header
func NodeHeader(col int) *Node {
    return nodeInit(0, col)
}
//Cover elimina il nodo dal grafo
func (n *Node) Cover() {
    if n.Head != nil {
        n.Up.Down = n.Down
        n.Down.Up = n.Up
```

```
        return
    }
    n.Left.Right = n.Right
    n.Right.Left = n.Left
}
```

Codice package terminal

```
package terminal
//fonte: Stackoverflow
import (
    "errors"
    "os"
    "os/exec"
    "runtime"
)

var clear map[string]func() //create a map for storing clear funcs

func init() {
    clear = make(map[string]func()) //Initialize it
    clear["linux"] = func() {
        cmd := exec.Command("clear") //Linux example, its tested
        cmd.Stdout = os.Stdout
        cmd.Run()
    }
    clear["darwin"] = func() {
        cmd := exec.Command("clear") //Linux example, its tested
        cmd.Stdout = os.Stdout
        cmd.Run()
    }
    clear["windows"] = func() {
        cmd := exec.Command("cmd", "/c", "cls") //Windows example, its tested
        cmd.Stdout = os.Stdout
        cmd.Run()
    }
}

func Clear() error {
    errUnsupported := errors.New("Your platform is unsupported! I can't clear terminal screen")
    value, ok := clear[runtime.GOOS] //runtime.GOOS -> linux, windows, darwin etc.
    if ok {                          //if we defined a clear func for that platform:
        value() //we execute it
```

```
    } else { //unsupported platform
        return errUnsupported
    }
    return nil
}
```

Codice package setting

```
package setting

//OutPathBT path dove scrivere i tempi dell'algoritmo Backtracking
const OutPathBT = "./tmp/solvingTimeBT.txt"

//OutPathDLX path dove scrivere i tempi dell'algoritmo Dancing Links
const OutPathDLX = "./tmp/solvingTimeDLX.txt"

//OutPathClues path dove scrivere il numero di indizi di ogni griglia
const OutPathClues = "./tmp/clues.txt"

//InputTest path dei file da passare come argomento alla funzione di test
var InputTest = []string{
    "./test/simple.txt",
    "./test/easy.txt",
    "./test/intermediate.txt",
    "./test/expert.txt",
    "./test/hardest.txt",
    "./test/fastTest.txt",
}
```

Codice package test

```
package test

import (
    "fmt"
    "log"
    "os"
    "strings"

    "../analyzetime"
    "../parse"
    "../setting"
    "../sudoku"
)

//Start avvia tutti i test e permette di aggiungerne di nuovi
func Start() {
    os.Remove(setting.OutPathBT)
    os.Remove(setting.OutPathDLX)
    os.Remove(setting.OutPathClues)
    var sel string
    fmt.Print("Vuoi eseguire i test standard o uno nuovo? [standard/nuovo]:")
    _, err := fmt.Scanf("%v", &sel)
    if err != nil {
        fmt.Println(err)
        log.Fatal(err)
    }
    sel = strings.ToLower(sel)
    switch sel {
    case "standard":
        for _, path := range setting.InputTest {
            testByLine(path, "#", ".")
            if len(setting.InputTest) == 1 {
                avgBt, err := analyzetime.Average(setting.OutPathBT)
                maxBt, err := analyzetime.Max(setting.OutPathBT)
                minBt, err := analyzetime.Min(setting.OutPathBT)
                avgDlx, err := analyzetime.Average(setting.OutPathDLX)
                maxDlx, err := analyzetime.Max(setting.OutPathDLX)
                minDlx, err := analyzetime.Min(setting.OutPathDLX)
                if err != nil {
                    fmt.Println(err)
```

```
            log.Fatal(err)
        }
        fmt.Printf("%v backtracking: average %.4f, max value %.4f,min value %.4f\n", path, avgBt, maxBt, minBt)
        fmt.Printf("%v dancing links: average %.4f, max value %.4f,min value %.4f\n", path, avgDlx, maxDlx, minDlx)
    } else {
        f, err := os.OpenFile(setting.OutPathBT, os.O_APPEND|os.O_WRONLY|os.O_CREATE, 0666)
        if err != nil {
            log.Fatal(err)
        }
        f2, err := os.OpenFile(setting.OutPathDLX, os.O_APPEND|os.O_WRONLY|os.O_CREATE, 0666)
        if err != nil {
            log.Fatal(err)
        }
        fc, err := os.OpenFile(setting.OutPathClues, os.O_APPEND|os.O_WRONLY|os.O_CREATE, 0666)
        if err != nil {
            log.Fatal(err)
        }
        _, err = fmt.Fprintf(f, "-------------------------\n")
        if err != nil {
            log.Fatal(err)
        }
        _, err = fmt.Fprintf(f2, "-------------------------\n")
        if err != nil {
            log.Fatal(err)
        }
        _, err = fmt.Fprintf(fc, "-------------------------\n")
        if err != nil {
            log.Fatal(err)
        }

        f.Close()
        f2.Close()
        fc.Close()
    }
}

case "nuovo":
    var pathin, pathout, delim, void string
    fmt.Print("Indicare il path del file da cui leggere i test: ")
    fmt.Scanf("%v", &pathin)
```

```
        fmt.Print("Indicare il path del file da cui leggere le soluzioni: ")
        fmt.Scanf("%v", &pathout)
        fmt.Print("Indicare il carattere che precede le righe da non leggere: ")
        fmt.Scanf("%v", &delim)
        fmt.Print("Indicare il valore che indica il numero mancante (di solito 0): ")
        fmt.Scanf("%v", &void)
        test(pathin, pathout, delim, void)
    default:
        fmt.Println("Errore nella selezione")
    }
}

//test legge i file di input ed output e esegue i test
func test(pathin, pathout, delim, void string) {
    gridIn := readTest(pathin, delim)
    gridOut := readTest(pathout, delim)
    for g := range gridIn {
        fmt.Println("test #:", g)
        test, err := sudoku.CreateFromString(gridIn[g], void)
        if err != nil {
            panic(err)
        }
        check, s1, s2 := sudoku.Solve(test)
        if !check {
            fmt.Println("Schema non risolto")
        }

        if strings.Compare(s1, s2) != 0 {
            fmt.Println("Due soluzioni trovate")
            fmt.Printf("%v\n%v\n", s1, s2)
        }
        compare(gridOut[g], s1)
    }
}

//readTest restituisce tutte le griglie lette dal file
func readTest(path string, delim string) [][]string {
    allGrid, err := parse.Parse(path, delim)
    var grid [][]string
    var single []string
```

```
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    for row := range allGrid {
        single = append(single, allGrid[row])
        if (row+1)%9 == 0 && row != 0 {
            grid = append(grid, single)
            single = nil
        }
    }
    return grid
}

//testByLine legge ed esegue tutte le griglie del file scritte come singole righe
func testByLine(pathin, delim, void string) {
    gridIn := readTestByLine(pathin, delim)
    for g := range gridIn {
        fmt.Printf("test #:%d ", g)
        test, err := sudoku.CreateFromString(gridIn[g], void)
        countClues(gridIn[g])
        if err != nil {
            panic(err)
        }
        check, s1, s2 := sudoku.Solve(test)
        if !check {
            fmt.Println("Schema non risolto")
        }

        if strings.Compare(s1, s2) != 0 {
            fmt.Println("Due soluzioni trovate")
            fmt.Printf("%v\n%v\n", s1, s2)
        }
        fmt.Println("risolto correttamente: ", s1)
    }
}

//readTestByLine legge e restituisce tutte le griglie di un file scritte come singole righe
func readTestByLine(path string, delim string) [][]string {
    allGrid, err := parse.Parse(path, delim)
```

```
    var grid [][]string
    var single []string
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    for row := range allGrid {
        str := strings.Split(allGrid[row], "")
        s := ""
        for i := range str {
            if i%9 == 0 && i != 0 {
                single = append(single, s)
                s = ""
            }
            s += str[i]
        }
        single = append(single, s) //append dell'ultima riga
        grid = append(grid, single)
        single = nil
    }
    return grid
}

//compare controllora se una griglia su più righe coincide con una su una singola riga
func compare(sol []string, str string) {
    var output string
    for i := range sol {
        output += sol[i]
    }
    if strings.Compare(output, str) == 0 {
        fmt.Println("Soluzione corretta")
    } else {
        fmt.Println("Soluzione errata")
        fmt.Println(str, "\n", output)
    }

}

//countClues conta il numero di indizi iniziali
func countClues(str []string) {
```

```
    c, _ := os.OpenFile(setting.OutPathClues, os.O_APPEND|os.O_WRONLY|os.O_CREATE, 0666)
    var empty int
    for i := range str {
        empty += strings.Count(str[i], ".")
    }
    fmt.Fprintf(c, "%d\n", 81-empty)
}
```