Lógica Computacional TP1

Realizado por: Miguel Gonçalves a90416 João Nogueira a87973

Universidade do Minho
Escola de Ciências

Exercicio 2

Da definição do jogo "Sudoku" generalizado para a dimensão N; o problema tradicional corresponde ao caso N=3. O objetivo do Sudoku é preencher uma grelha de $N^2 \times N^2$ com inteiros positivos no intervalo 1 até N^2, satisfazendo as seguintes regras

1. **Cada inteiro no intervalo 1 até N^2 ocorre só uma vez em cada coluna, linha e secção $N \times N$.**
2. **No início do jogo uma fração $0 \leq \alpha < 1$ das N^4 casas da grelha são preenchidas de forma consistente com a regra anterior.**

a) Construir um programa para inicializar a grelha a partir dos parâmetros N e α

b) Construir soluções do problema para as combinações de parâmetros $N \in \{3,4,5,6\}$ e $\alpha \in \{0.0, 0.2, 0.4, 0.6\}$. Que conclusões pode tirar da complexidade computacional destas soluções.

Obrigações

1. **Cada inteiro no intervalo 1 até N^2 ocorre só uma vez em cada coluna, linha e secção $N \times N$:</p>**

$$\forall_l \cdot \forall_c \cdot \forall_q$$
$$\sum |n|_x = 1$$

1. **No início do jogo uma fração $0 \leq \alpha < 1$ das N^4 casas da grelha são preenchidas de forma consistente com a regra anterior:**

$$\forall_l \cdot \forall_c \cdot \forall_q$$
$$\sum |n|_A$$
$$= N^4 \times \alpha$$

a) $\sum_{v=1}^{N^2} x_{vrc} = 1$

b) $\sum_{r=Np-2}^{Np} \sum_{c=Nq-2}^{Nq} x_{vrc} = 1$

$r, c, v \in 1, N^2 \land p, q \in 1, N$

onde l representa linhas, c representa colunas, q representa quadrados, $A \in \{1, N^4\}^*$ e $n \in \{1, N^2\}$.

Resolver com o SCIP

In [ ]:

```
!pip install ortools
```

In [ ]:

```
import random
import itertools
import math
from ortools.linear_solver import pywraplp
import pandas as pd

def inicia_sudoku_scip(n,alpha):

    #Para gerar o sudoku vamos preencher algumas casas aleatoriamente, mas respeitando as regras do sudoku
    #Completamos o sudoku e no fim apagamos o numero pretendido de celulas e desta maneira sabemos que o sudoku
    #tem soluçao
    if alpha < 0 or alpha > 1:
        return "Error"  #O alpha tem de estar sempre entre 0 e 1
    hints = 0
    hints = math.ceil((n**4) * alpha) #numero de casas que tem de preencher, é sempre arredondado para cima
    tabuleiro = [[ 0 for _ in range(0,n**2)] for _ in range(0,n**2)]
    ocupados = []
    pistas =  math.ceil((n**4) * 0.2)

    while(pistas>0):
        linha = random.randint(0,n**2-1)
        coluna = random.randint(0,n**2-1)

        if (linha,coluna) not in ocupados:
            ocupados.append((linha,coluna))
            posto = False
            while(not posto):
                numeroapor = random.randint(1,n**2)
                if numerovalido(numeroapor,linha,coluna,tabuleiro):
                        tabuleiro[linha][coluna] = numeroapor
                        posto = True
                else:
                    posto = False
            pistas -=1

    tabuleirofinal = sudoku_solver_scip(tabuleiro)
    removido = []

    while(hints>0):
        linha = random.randint(0,n**2-1)
        coluna = random.randint(0,n**2-1)
        if (linha,coluna) not in removido:
            removido.append((linha,coluna))
            hints -=1

    for l in range (0,n**2):
        for c in range(0,n**2):
            if (l,c) not in removido:
                tabuleirofinal[l][c] = 0
    solver = pywraplp.Solver.CreateSolver('SCIP')
    printabonitoscip(tabuleirofinal)

    return tabuleirofinal


def numerovalido(n,linha,coluna,grelha):

    for i in range(0,len(grelha)):
```

```
        if i==coluna:
            for j in range(0,len(grelha)):
                if grelha[j][coluna]==n:
                    return False

    for i in range(0,len(grelha)):
        if i==linha:
            if n in grelha[linha]:
                return False

    profundidade = 1
    quadrado = emquequadradoesta(linha,coluna,len(grelha))

    offsets = list(itertools.product(range(0,int(math.sqrt(len(grelha)))), range(0, int(
math.sqrt(len(grelha))))))
    for r in range(0, len(grelha), int(math.sqrt(len(grelha)))):
        for c in range(0, len(grelha), int(math.sqrt(len(grelha)))):
            agrupar = []
            for dy, dx in offsets:
                agrupar.append(grelha[r+dy][c+dx])
            if profundidade == quadrado:
                if n in agrupar:
                    return False
            profundidade += 1
    return True


def emquequadradoesta(linha,coluna,tamanho):
    ret = 1
    while linha>int(math.sqrt(tamanho))-1:
        linha -= int(math.sqrt(tamanho))
        ret *= 4

    while coluna>int(math.sqrt(tamanho))-1:
        coluna -= int(math.sqrt(tamanho))
        ret += 1

    return ret

def printabonitoscip(tabuleiro):
    for r in range(0,len(tabuleiro)):
        for c in range(0,len(tabuleiro)):
            #printa o valor correto atribuido à celula em questao
            v = tabuleiro[r][c]
            print(v, end=' ')
            if (c+1) % int(math.sqrt(len(tabuleiro))) == 0:
                print('  ', end='')
        print()
        if (r+1) % int(math.sqrt(len(tabuleiro))) == 0:
            print()
    print()



def sudoku_solver_scip(grelha):

    cell_size = int(math.sqrt(len(grelha)))
    line_size = cell_size**2
    line = list(range(0, line_size))
    cell = list(range(0, cell_size))
    grid = {}
    initial_grid = pd.DataFrame(grelha)
    solver = pywraplp.Solver.CreateSolver('SCIP')

    # Variaveis
    for i in range(0,len(grelha)):
        for j in range(0,len(grelha)):
            for k in range(0,len(grelha)):
                grid[i, j, k] = solver.BoolVar(f"grid[{i},{j},{k}]")

    # Constraints
    for i in range(0,len(grelha)):
```

```python
        for j in range(0,len(grelha)):
            # Cada numero nao se pode repetir na sua linha
            solver.Add(solver.Sum(grid[i, k, j] for k in range(0,len(grelha))) == 1)
            # Cada numero nao se pode repetir na sua coluna
            solver.Add(solver.Sum(grid[k, i, j] for k in range(0,len(grelha))) == 1)
            # Apenas um valor em cada quadrado
            solver.Add(solver.Sum(grid[i, j, k] for k in range(0,len(grelha))) == 1)

    # Garantir que todos os valores sao diferentes no seu quadrado
    for i in range(0,int(math.sqrt(len(grelha)))):
        for j in range(0,int(math.sqrt(len(grelha)))):
            for k in range(0,int(math.sqrt(len(grelha)))):
                solver.Add(
                    solver.Sum(grid[i * cell_size + di, j * cell_size + dj, k] \
                        for di in cell for dj in cell) == 1
                )

    # Adicionar as pistas que temos
    for i in range(0,len(grelha)):
        for j in range(0,len(grelha)):
            value = initial_grid.loc[i, j]
            if value:
                solver.Add(grid[i, j, value - 1] == 1)

    status = solver.Solve()
    tabuleirofinal = [[ None for _ in range(0,len(grelha))] for _ in range(0,len(grelha)
)]
    if status == pywraplp.Solver.OPTIMAL:
        for i in line:
            for j in line:
                # Vamos por os valores no sitio certo
                tabuleirofinal[i][j] = [grid[i, j, k].solution_value() for k in range(0,
len(grelha))].index(1) + 1
        #printabonitoscip(tabuleirofinal)
    else:
        print("Imp")
    return tabuleirofinal

print("Sudoku por preencher")
grelha = inicia_sudoku_scip(3,0.4)
resolvido = sudoku_solver_scip(grelha)
print("Sudoku Resolvido")
printabonitoscip(resolvido)
```

```
Sudoku por preencher
0 0 0   0 2 0   0 3 8
0 0 6   0 5 0   1 7 0
0 4 0   0 0 0   0 9 6

0 8 0   0 0 6   0 0 0
0 3 7   2 0 0   5 6 0
4 5 0   0 9 0   2 1 0

0 1 0   0 0 0   0 0 4
8 7 9   0 4 0   6 0 3
6 0 0   9 0 5   0 0 0


Sudoku Resolvido
7 6 1   5 2 9   4 3 8
3 9 6   4 5 8   1 7 2
2 4 5   3 7 1   8 9 6

9 8 2   7 1 6   3 4 5
1 3 7   2 8 4   5 6 9
4 5 8   6 9 3   2 1 7

5 1 3   8 6 7   9 2 4
8 7 9   1 4 2   6 5 3
6 2 4   9 3 5   7 8 1
```

Resolver com o z3

In [ ]:

```
!pip install z3-solver
```

In [ ]:

```
import z3

def inicia_sudoku(n,alpha):

    #Estrategia é criar um sudoku resolvido e apagar celulas
    #A grelha vai ser uma lista com listas... O numero de listas vai ser n^2 e o tamanho
de cada lista é n^2

    count = 0
    colunasocupadas = []
    dic = {}
    preenchidos = 0

    if alpha < 0 or alpha > 1:
        return "Error"  #O alpha tem de estar sempre entre 0 e 1
    hints = 0
    hints = math.ceil((n**4) * alpha) #numero de casas que tem de preencher, é sempre ar
redondado para cima


    s = z3.Solver()

    tabuleiroaux = [[ None for _ in range(0,n**2)] for _ in range(0,n**2)]

    for r in range(0,n**2):
        for c in range(0,n**2):
            v = z3.Int('c_%d_%d' % (r, c))
            tabuleiroaux[r][c] = v

    while(count<hints):
        #Todas as linhas têm de ter pelo menos uma pista
        for line in range(n**2):
            aux = random.randint(1,n**2)
            if aux-1 not in colunasocupadas:
                dic[line,aux-1] = 1
                preenchidos+=1
                colunasocupadas.append(aux-1)
            else:
                if len(colunasocupadas)<9:
                    dic[line,aux-1] = 1
                    preenchidos+=1
                    colunasocupadas.append(aux-1)
                else:
                    aux = random.randint(1,n**2)
            count += 1

    faltapreencher = hints-preenchidos

    #Todas as colunas têm de ter pelo menos uma pista
    for k in range(n**2):
        if k not in colunasocupadas and faltapreencher!=0:
            aux = random.randint(1,n**2)
            dic[aux-1,k] = 1
            faltapreencher -= 1


    while(faltapreencher>0):
        line = random.randint(1,n**2)
        colum = random.randint(1,n**2)
        if (line,colum) not in dic:
            dic[line,colum] = 1
```

```
            faltapreencher -= 1
        else:
            linha = random.randint(1,n**2)
            coluna = random.randint(1,n**2)

    #Vamos adicionar as restriçoes do sudoku
    restricoes(s, tabuleiroaux)
    status = s.check()

    print(status)

    tabuleirofinal = []
    if status == z3.sat:
        #Se for satisfazivel entao vamos printar de maneira bonita
        tabuleirofinal = apaga(s,tabuleiroaux,dic)
        printar(tabuleirofinal)
    return tabuleirofinal

def apaga(s,tabuleiroaux,dic):
    tabuleirofinal = [[ None for _ in range(0,len(tabuleiroaux))] for _ in range(0,len(t
abuleiroaux))]
    m = s.model()
    for r in range(0,len(tabuleiroaux)):
        for c in range(0,len(tabuleiroaux)):
            #printa o valor correto atribuido à celula em questao
            if (r,c) in dic:
                v = m.evaluate(tabuleiroaux[r][c])
                tabuleirofinal[r][c] = v
            else:
                tabuleirofinal[r][c] = 0

    return tabuleirofinal

def printar(tabuleirofinal):
    for r in range(0,len(tabuleirofinal)):
        for c in range(0,len(tabuleirofinal)):
            v = tabuleirofinal[r][c]
            print(v, end=' ')
            if (c+1) % int(math.sqrt(len(tabuleirofinal))) == 0:
                print('  ', end='')
        print()
        if (r+1) % int(math.sqrt(len(tabuleirofinal))) == 0:
            print()
    print()

def restricoes(s, tabuleiroaux):

    #Todos os valores têm de estar entre 1 e n^2
    for r in range(0,len(tabuleiroaux)):
        for c in range(0,len(tabuleiroaux)):
            v = tabuleiroaux[r][c]
            s.add(v >= 1)
            s.add(v <= len(tabuleiroaux))

    #Todas as celulas da mesma linha têm de ter valores diferentes
    for r in range(0,len(tabuleiroaux)):
        s.add(z3.Distinct(tabuleiroaux[r]))

    #Todas as celulas da mesma coluna têm de ter valores diferentes
    for c in range(0,len(tabuleiroaux)):
        col = [tabuleiroaux[r][c] for r in range(0,len(tabuleiroaux))]
        s.add(z3.Distinct(col))

    #Todas as celulas em nxn têm de ter valores diferentes.
    #Vamos dividir o tabuleiro em sublistas de nxn tamanho

    offsets = list(itertools.product(range(0,int(math.sqrt(len(tabuleiroaux)))), range(0
, int(math.sqrt(len(tabuleiroaux))))))
    for r in range(0, len(tabuleiroaux), int(math.sqrt(len(tabuleiroaux)))):
        for c in range(0, len(tabuleiroaux), int(math.sqrt(len(tabuleiroaux)))):
            agrupar = []
            for dy, dx in offsets:
```

```
            agrupar.append(tabuleiroaux[r+dy][c+dx]) #percorre os quadrados todos
        s.add(z3.Distinct(agrupar))
```

In [ ]:

```
def convertepistas(grelha):
    dic = {}
    ret = {}
    for n in range(len(grelha)):
        for k in range(len(grelha)):
            dic[n,k]=grelha[n][k]
    for (a,b) in dic:
        if dic[a,b]==0:
            dic[a,b]=0
        else:
            ret[a,b] = dic[a,b]
    return ret

def solve_sudoku(g):
    #vamos passar os valores que estao no tabuleiro para um dicionario
    pistas = convertepistas(g)
    s = z3.Solver()

    #É criado um tabuleiro no qual vai ser resolvido o sudoku

    for r in range(0,len(g)):
        for c in range(0,len(g)):
            v = z3.Int('c_%d_%d' % (r, c))
            g[r][c] = v
            #Se a celula em questao estiver no dicionario (sao as pistas) entao vamos sub
stituir essa celula pelo valor
            #proposto no dicionario
            if (r, c) in pistas:
                s.add(v == pistas[(r, c)])

    #Vamos adicionar as restriçoes do sudoku
    restricoes(s, g)
    status = s.check()

    print(status)
    if status == z3.sat:
        #Se for satisfazivel entao vamos printar de maneira bonita
        printabonito(s,g)

def printabonito(s,g):
    m = s.model()
    for r in range(0,len(g)):
        for c in range(0,len(g)):
            #printa o valor correto atribuido à celula em questao
            v = m.evaluate(g[r][c])
            print(v, end=' ')
            if (c+1) % int(math.sqrt(len(g))) == 0:
                print('  ', end='')
        print()
        if (r+1) % int(math.sqrt(len(g))) == 0:
            print()
    print()


grelha = inicia_sudoku(3,0.2)
solve_sudoku(grelha)
```

```
sat
0 0 0   0 0 9   0 0 0
0 0 0   4 5 0   0 0 8
0 0 0   6 0 0   0 4 0

0 4 0   0 7 0   0 0 0
0 0 1   0 0 0   0 5 0
5 0 0   0 0 0   0 0 0

0 0 0   0 0 0   1 0 0
```

```
0 9 7   3 0 0   0 8 0
0 0 0   0 0 4   0 0 0


sat
3 7 4   8 2 9   5 6 1
1 2 6   4 5 7   3 9 8
9 8 5   6 1 3   7 4 2

8 4 2   9 7 5   6 1 3
7 6 1   2 3 8   9 5 4
5 3 9   1 4 6   8 2 7

4 5 8   7 9 2   1 3 6
2 9 7   3 6 1   4 8 5
6 1 3   5 8 4   2 7 9
```

Conclusões

Podemos concluir que quanto maior for o n, mais tempo irá demorar para completar a execução das funções uma vez que o numero de opções a serem avaliadas são maiores quanto maior for o n. Com isto, podemos afirmar que quanto maior for o n, maior irá ser a complexidade computacional. Por outro lado, quanto maior for o α menor é o tempo de execução da função que resolve o sudoku. Isto acontece uma vez que quanto maior for o α mais "pistas" vão ser dadas no tabuleiro inicial e deste modo irão haver menos casos a serem avaliados.

Observações

O tempo de execução da inicialização da grelha é alto uma vez que primeiro é gerado o sudoku completo e só depois apagamos algumas celulas