Trabalho 1

Grupo 06

- Tomás Vaz de Carvalho Campinho A91668
- Miguel Ângelo Alves de Freitas A91635

Problema 2

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.

   a. Cada inteiro no intervalo,1 até N^2 ocorre só uma vez em cada coluna, linha e secção N times N.

   b. No início do jogo uma fração 0 <=a < 1 das N^4 casas da grelha são preenchidas de forma consistente com a regra anterior.

   2.1 Construir um programa para inicializar a grelha a partir dos parâmetros N e alpha

   2.2 Construir soluções do problema para as combinações de parâmetros in {3,4,5,6}e, a em {0.0,0.2,0.4,0.6} . Que conclusões pode tirar da complexidade computacional destas soluções.

```
!pip install ortools
```

```
from ortools.linear_solver import pywraplp
import networkx as nx
```

Ponto de partida

1. Quando pensamos em fazer o sudoku imaginamos que o sudoku seria uma matriz uma vez que é uma tabela e em cada celula (x,y) iriamos ter o valor dessa tabela. Então a seguinte função vai gerar um sudoku sempre diferente e válido. Para a estratégia de gerar o sudoku basicamente fizemos por escolha aleatoria, primeiramente geramos uma matriz preenchida com tudo a zeros.

```
import random
import networkx as nx
```

```
def Sudoku(m, a):
    n = m*m
    a = round(n*n*a)
    tabela = [[0 for x in range(n)] for y in range(n)]

    for i in range(n):
        for j in range(n):
            tabela[i][j] = 0

    for i in range(a):
        linha = random.randrange(n)
        coluna = random.randrange(n)
        num = random.randrange(1,n+1)
        while(not sitiovalido(tabela,linha,coluna,num,n) or tabela[linha][coluna] != 0):
            linha = random.randrange(n)
            coluna = random.randrange(n)
            num = random.randrange(1,n+1)
        tabela[linha][coluna]= num;

    print(tabela)
```

2. Basicamente vai escolher um número aleatorio e vai preencher num espaço aleatório e depois vai verificar se é válido nesse espaço e também num sitio que ainda não foi preenchido. Vai comparar com o resto da tabela ver se se é valido, caso seja válido dá return true. As restrições que fizemos foram as seguintes:

a. ver se esse numero é igual na tabela e na linha.

b. ver se é valido no quadrado que se encontra.

```
def sitiovalido(tabela,linha,coluna,num,n):
    valid = True
    j= n//2
    for x in range(n):
        if (tabela[x][coluna] == num):
            valid = False
    for y in range(n):
        if (tabela[linha][y] == num):
            valid = False
    linhasection = linha // j
    colunasection = coluna // j
    for x in range(j):
        for y in range(j):
          #vê se a secção é valida
            if(tabela[linhasection*j + x][colunasection*j + y] == num):
                valid = False
    return valid
```

```
Sudoku(4,0.2)
```

```
[[0, 0, 0, 0, 0, 4, 0, 3, 0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 7, 0, 0, 0, 12, 0, 0,
```

Solver

1. Para o exercicio 2 alinea 2.2 nós optamos por obter um sudoku feito e resolvido por grafos, assim não temos o problema de ter que inverter a matriz para um grafo e depois colorir e voltar a passar para matriz.

2. Sendo assim optamos pela seguinte função que testa se é possível colorir um grafo com k cores usando o OR-Tools com o solver BOP que é um versão mais rápida do SCIP, usada em aula. Se for possível, deve guardar a coloração no grafo (no atributo color). Usamos um dicionário x para guardar as varáveis, sendo que em x[v][c] será armazenada a variável $x_{v,c}$.

```
def ip_color_op(graph,k):
    # criar solver
    solver = pywraplp.Solver('BOP', pywraplp.Solver.BOP_INTEGER_PROGRAMMING)
    #criar dicionario de variaveis x{i,j}
    x = {}
    for i in graph:
      x[i] = {}
      for j in range(k):
        x[i][j] = solver.BoolVar('x[%i][%i]' % (i,j))

    # vertices adjacentes tem cores diferentes
    for o in graph:
      for d in graph[o]:
        for j in range(k):
          solver.Add(x[o][j] + x[d][j] <= 1)

    # Manter o que ja tem cor
    for v in graph:
      if 'color' in graph.nodes[v]:
        solver.Add(x[v][graph.nodes[v]['color']] == 1)


    for i in graph:
      solver.Add(sum([x[i][j] for j in range(k)]) == 1)  # ou solver.Add(sum(list(x[i].values

    # invocar solver e colorir o grafo

    status = solver.Solve()
    if status == pywraplp.Solver.OPTIMAL:
        for i in graph:
            for j in range(k):
```

```
                if round(x[i][j].solution_value())==1:
                    graph.nodes[i]['color'] = j
        return True
    else:
        return False


def sudoku(N):
  graph = nx.Graph()

  # nodos
  for i in range(1,(N**4)+1):
    graph.add_node(i)

  # colunas
  for i in range(1,(N**4)+1):
    for j in range(i+N**2,(N**4)+1,N**2):
      graph.add_edge(i,j)

  # linhas
  lim = N**2
  for casa in range(1,(N**4)+1):
    for k in range(casa+1,lim+1):
      graph.add_edge(casa, k)
    if casa % N**2 == 0:
      lim = lim+N**2

  # quadrados
  dic = {}
  for i in range(N**2):
    dic[i] = []

  k = 0
  p = 1
  l = 0
  for i in range(1,N**4+1):
    dic[l + k].append(i)
    if i % N == 0:
      k = (k+1) % N
    if i % N**2 == 0:
      p+=1
      if p > N:
        l+=N
        p=1

  for lista in dic.values():
    for i in lista:
      for j in lista:
        if i != j:
          graph.add_edge(i,j)

  #print(graph.edges())
```

```
  assert ip_color_op(graph, N**2)
  # draw_with_colors(graph)

  return graph
```

3. Para fazer print ao sudoku no inicio tivemos algumas dúvidas de como poderiamos mostrar o sudoku, desta forma optamos por separar por quadrados assim fica mais visivel à sua visualização

```
def print_sudoku(graph, N):
  num = 1
  for i in range(N**2):
    for j in range(N):
      print(" ", end ="")
      for k in range(N):
        if 'color' in graph.nodes[num]:
          print("%02d" % (graph.nodes[num]['color']+1), end =" ")
        else:
          print("..", end =" ")
        num+=1
      if j != N-1:
        print(" ", end ="")
    print("\n", end ="")
    if (i+1) % N == 0 and i != N**2-1:
      for y in range(N):
        for x in range((3*N)+1):
          print(" ", end ="")
        if y != N-1:
          print(" ", end ="")
        else:
          print("\n", end ="")
```

Exemplo de Sudoku gerado para N=3

```
import timeit
t = timeit.timeit(lambda: print_sudoku(sudoku(3),3), number = 1)

print("Tempo de execução:",t)
```

```
      03 04 06   01 08 05   02 07 09
      01 02 05   09 03 07   08 04 06
      08 07 09   02 04 06   01 03 05

      05 06 03   04 01 02   07 09 08
      04 01 02   08 07 09   05 06 03
```

```
 07 09 08   05 06 03   04 01 02

 06 05 04   03 02 01   09 08 07
 02 03 01   07 09 08   06 05 04
 09 08 07   06 05 04   03 02 01
Tempo de execução: 0.2685621489999903
```

Exemplo de Sudoku gerado para N=4

```
t = timeit.timeit(lambda: print_sudoku(sudoku(4),4), number = 1)
print("Tempo de execução:",t)
```

```
 05 02 03 04   01 06 07 08   09 10 11 12   13 14 15 16
 01 06 07 08   05 02 03 04   13 14 15 16   09 10 11 12
 09 10 11 12   13 14 15 16   01 02 03 04   05 06 07 08
 13 14 15 16   09 10 11 12   05 06 07 08   01 02 03 04

 02 01 04 03   06 05 08 07   10 09 12 11   14 13 16 15
 06 05 08 07   02 01 04 03   14 13 16 15   10 09 12 11
 10 09 12 11   14 13 16 15   02 01 04 03   06 05 08 07
 14 13 16 15   10 09 12 11   06 05 08 07   02 01 04 03

 03 04 01 02   07 08 05 06   11 12 09 10   15 16 13 14
 07 08 05 06   03 04 01 02   15 16 13 14   11 12 09 10
 11 12 09 10   15 16 13 14   03 04 01 02   07 08 05 06
 15 16 13 14   11 12 09 10   07 08 05 06   03 04 01 02

 04 03 02 01   08 07 06 05   12 11 10 09   16 15 14 13
 08 07 06 05   04 03 02 01   16 15 14 13   12 11 10 09
 12 11 10 09   16 15 14 13   04 03 02 01   08 07 06 05
 16 15 14 13   12 11 10 09   08 07 06 05   04 03 02 01
Tempo de execução: 2.5104190550000567
```

Exemplo de Sudoku gerado para N=5

```
t = timeit.timeit(lambda: print_sudoku(sudoku(5),5), number = 1)
print("Tempo de execução:",t)
```

```
 03 14 06 07 16   09 19 11 24 23   04 22 12 17 08   01 15 13 10 18   25 20 02 21 05
 24 15 17 09 08   02 14 03 16 06   25 05 20 11 23   19 21 07 12 04   13 01 10 22 18
 02 05 20 13 01   18 10 08 21 12   06 09 15 24 07   25 22 03 23 11   16 14 04 19 17
 22 10 12 11 21   25 04 20 13 01   19 02 03 16 18   14 17 08 24 05   23 09 07 15 06
 18 19 23 25 04   05 15 17 22 07   10 14 13 21 01   06 16 09 20 02   11 03 08 12 24

 06 09 11 21 19   24 08 15 17 20   07 01 04 25 16   18 14 12 22 10   03 02 13 05 23
 05 02 01 15 17   16 06 07 04 14   23 08 18 20 10   03 11 25 13 09   22 12 21 24 19
 12 04 07 24 14   11 13 23 09 05   17 03 22 19 21   02 06 16 15 20   08 18 25 01 10
 20 03 10 23 18   01 21 12 25 22   14 13 02 05 06   04 07 24 08 19   17 11 09 16 15
 13 16 25 08 22   03 18 10 02 19   09 15 24 12 11   17 05 23 21 01   20 07 06 04 14
```

```
 15 25 16 14 09   07 17 22 12 21   13 20 10 04 02   23 03 01 06 24   18 19 05 11 08
 11 20 05 04 24   10 03 06 23 16   12 18 07 22 09   08 13 19 02 15   14 17 01 25 21
 07 18 13 10 12   19 09 25 14 08   11 17 06 01 15   16 04 22 05 21   02 24 20 23 03
 08 23 22 19 06   20 02 01 05 15   16 21 14 03 24   11 09 17 18 25   10 04 12 07 13
 01 17 21 02 03   04 11 24 18 13   05 19 23 08 25   10 20 14 07 12   09 22 15 06 16

 14 11 24 22 05   15 12 04 19 17   02 07 08 18 03   09 25 20 16 06   21 13 23 10 01
 09 12 19 16 10   21 07 18 03 02   15 25 01 13 20   24 23 05 04 14   06 08 11 17 22
 17 01 04 06 25   13 20 16 08 10   24 23 05 14 12   21 18 02 11 22   19 15 03 09 07
 21 07 03 18 02   23 25 05 06 09   22 11 17 10 04   13 19 15 01 08   24 16 14 20 12
 23 08 15 20 13   22 24 14 01 11   21 06 16 09 19   07 12 10 17 03   04 05 18 02 25

 04 21 08 17 23   14 05 19 07 24   18 12 09 15 13   20 02 06 25 16   01 10 22 03 11
 16 24 09 01 15   12 22 02 11 04   03 10 25 06 17   05 08 21 14 13   07 23 19 18 20
 10 22 18 05 07   06 16 21 20 03   08 04 19 23 14   12 01 11 09 17   15 25 24 13 02
 19 13 02 12 11   08 01 09 10 25   20 16 21 07 22   15 24 18 03 23   05 06 17 14 04
 25 06 14 03 20   17 23 13 15 18   01 24 11 02 05   22 10 04 19 07   12 21 16 08 09
Tempo de execução: 16.44587977900005
```

Conclusões

As conclusões que o nosso grupo conseguiu tirar em relação à resolução deste sudoku foi que quanto maior for o n e maior for o alpha o tempo de execução do solver vai crescer exponencialmente.

Sempre que aumentavamos o numero de células resolvidas o sudoku demorava mais tempo a gerar pois ele não consegue verificar e preencher tudo ao mesmo tempo em "N" tempo. Aliás se tentassemos gerar um sudoku preenchido ele iria demorar muito tempo.

Quanto à primeira parte observamos que parte do problema é para cada Sudoku 9x9 válido tem os dígitos 1-9 uma vez em cada linha, e para cada tabela que gerar com essa propriedade encontraramos 1.8e27 que não tem. Isso é pouco mais do que a batata do meu computador pode suportar.

A maioria das estratégias que conheço recaem em uma breadth or depth first search. Ou seja se quisessemos melhorar o tempo de execução do primeiro código teriamos que:

- Colocar cada dígito em ordem (1s, depois 2s, ...). A maioria das maneiras de fazer isso termina em Sudokus válidos, ao passo que, especialmente para sudokus maiores, a construção de um quadrado por vez geralmente termina em Sudokus inválidos.

Que por sua vez foi o que tentamos fazer na segunda parte do trabalho.´

Outro tipo de estratégia que poderiamos ter utilizado e era sugerido em muitos sites pela internet era gerar um sudoku válido e apagar e resolver de novo "https://liorsinai.github.io/coding/2020/07/27/sudoku-solver.html". Neste caso Backtracking é um algoritmo que tenta recursivamente soluções potenciais e remove aquelas que não funcionam.

Sendo assim conlcuimos que O (n ^ m) onde n é o número de possibilidades para cada quadrado (ou seja, 9 no Sudoku clássico) em é o número de espaços que estão em branco.

Se houver três espaços em branco, vamos trabalhar com n possibilidades para o primeiro espaço em branco. Cada uma dessas possibilidades produzirá um puzzle com dois espaços em branco que possui n ^ 2 possibilidades. Concluindo que este algoritmo a sua complexidade vai ser sempre exponencial.

Cada nível do gráfico representa as opções para um único quadrado. A profundidade do gráfico é o número de quadrados que precisam ser preenchidos. Com um fator de ramificação de n e uma profundidade de m, encontrar uma solução no gráfico tem um desempenho de pior caso de O (n ^ m).