Universidade do Minho

Licenciatura em Ciências da Computação

Trabalho 1

Lógica Computacional 2019/2020

Docente: Manuel José Valença

Grupo 11

Ilda Durães	A78195
João Cerqueira	A65432

29 de Outubro de 2019

Conteúdo

1 Introdução

Este trabalho tem como objetivo determinar uma solução para dois problemas, o problema do *Sudoku* e um problema num sistema de tráfego. Neste sentido, este trabalho visa definir restrições para resolver os problemas, utilizando o *solver SCIP* com suporte *PySCIPOpt*.

No presente relatório cada problema terá uma descrição, a concepção (isto é, quais os objetivos e as restrições definidas para a resolução do problema) e a resolução do problema, onde constam os algoritmos, testes realizados e resultados obtidos.

2 Sudoku

2.1 Descrição do problema

O *Sudoku* é um jogo baseado na colocação lógica de números, criado por *Howard Garns*, um arquiteto americano. O objetivo do jogo é a colocação de números de 1 a N em cada uma das células vazias numa grade de N^2xN^2, constituída por NxN subgrades chamadas regiões.

O problema tradicional corresponde ao caso N=3, todavia existem versões do jogo com outras dimensões, nomeadamente N=4,5,6.

O quebra-cabeça contém algumas pistas iniciais, que são números inseridos em algumas células, de maneira a permitir uma indução ou dedução dos números em células que estejam vazias. Cada coluna, linha e região só pode ter cada um dos números de 1 a N.

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

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

Figura 1: Exemplo do jogo Sudoku antes e após a sua resolução, respectivamente

2.2 Concepção

2.2.1 Restrições

Para este problema, assumimos que os valores i e j percorrem a matriz da seguinte forma:

- **i:** indíce da linha do sudoku
- **j:** indíce da coluna do sudoku
- **val:** o valor da célula de 1 a N^2

i. Cada célula só pode ter um e um só valor, entre 1 a N^2.
Esta restrição pode expressar-se da seguinte forma:

$$\forall_{i<N^2}.\forall_{j<N^2}. \quad \big(\sum_{val<N^2} x_{i,j,val}\big) = 1$$

ii. Cada linha terá N^2 espaços com os valores de 1 até N^2.
Esta restrição pode expressar-se da seguinte forma:

$$\forall_{i<N^2}.\forall_{val<N^2}. \quad \big(\sum_{j<N^2} x_{i,j,val}\big) = 1$$

iii. Cada coluna terá N^2 espaços com os valores de 1 até N^2.
Esta restrição pode expressar-se da seguinte forma:

$$\forall_{j<N^2}.\forall_{val<N^2}. \quad \big(\sum_{i<N^2} x_{i,j,val}\big) = 1$$

iv. Cada quadrado terá N^2 espaços com os valores de 1 até N^2.
Esta restrição pode expressar-se da seguinte forma:

$$\forall_{i<N}.\forall_{j<N}.\forall_{val<N}. \quad \big(\sum_{linha<N, coluna<N} x_{linha+N*i, coluna+N*j, val}\big) = 1$$

2.3 Resolução do problema

2.3.1 Algoritmos

A função *sudokuSolver* apresentada posteriormente, irá receber como parâmetros uma lista com o *sudoku* não resolvido e um valor inteiro N e devolver o *sudoku* devidamente resolvido.

```
from pyscipopt import Model, quicksum

def sudokuSolver(sudoku, n):

    # criar as variveis
    m = Model();
    x = {};

    # preencher com valores iniciais:
    for i in range(n**2):
        x[i] = {}
        for j in range(n**2):
            x[i][j] = {}
            for val in range(n**2):
                x[i][j][val] = m.addVar(str(i) + str(j) + str(val),
                    vtype = 'B')


    for i in range(n**2):
        for j in range(n**2):
            if sudoku[j + (n**2) * i] != 0:
                m.addCons(x[i][j][sudoku[j + (n**2) * i] - 1] == 1)

    # a matriz  lida linha a linha, coluna a coluna. Logo, i = linha,
        j = coluna
```

```python

    # restricao i.
    # cada clula  no  pode ter mais do que 1 valor de 1 a N x N.
    for i in range(n**2):
        for j in range(n**2):
            m.addCons(quicksum(x[i][j][val] for val in range(n**2)) == 1)

    # restricao ii.
    # cada linha ter N x N espaos com os valores de 1 a N x N (sem repetidos).
    for i in range(n**2):
        for val in range(n**2):
            m.addCons(quicksum(x[i][j][val] for j in range(n**2)) == 1)

    # restricao iii.
    # cada coluna ter N  espaos  com os valores de 1 a N x N (sem repetidos).
    for j in range(n**2):
        for val in range(n**2):
            m.addCons(quicksum(x[i][j][val] for i in range(n**2)) == 1)

    # restricao iv.
    # cada  seco  N x N  ter  os valores de 1 a N x N (sem repetidos).
    for i in range(n):
        for j in range(n):
            for val in range(n**2):
                m.addCons( quicksum( x[linha+n*i][coluna+n*j][val] for coluna in range(n) for linha in range(n)) == 1 )

    m.optimize()

    print(m.getStatus())

    if m.getStatus() == 'optimal':
        solucao = {}
        for i in range(n**2):
            solucao[i] = {}
            out = ''
            for j in range(n**2):
                solucao[i][j] = {}
                for val in range(n**2):
                    if m.getVal(x[i][j][val]) == 1:
                        solucao[i][j] = val + 1
                out += str(solucao[i][j]) + ' '
            print(out)
```

2.3.2 Testes realizados e resultados obtidos

Esta secção visa apresentar exemplos de testes realizados e os respectivos resultados ou conclusões obtidas.

- N=3

Passamos à função *sudokuSolver* os parâmentros n=3 e o seguinte *sudoku*:

```
init3 = [0, 0, 0, 0, 3, 0, 0, 1, 0,
        0, 5, 7, 0, 0, 2, 4, 0, 0,
        1, 6, 0, 9, 0, 5, 0, 0, 0,
        9, 0, 0, 0, 0, 0, 0, 0, 4,
        4, 0, 1, 8, 0, 3, 2, 0, 5,
        6, 0, 0, 0, 0, 0, 0, 0, 8,
        0, 0, 0, 3, 0, 7, 0, 4, 2,
        0, 0, 2, 4, 0, 0, 6, 8, 0,
        0, 8, 0, 0, 2, 0, 0, 0, 0]
```

E obtemos como resultado:

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

- N=4

Passamos à função *sudokuSolver* os parâmentros n=3 e o seguinte *sudoku*:

```

init4 = [0, 2, 14, 0, 0, 0, 16, 4, 0, 0, 0, 1, 0, 0, 5, 0,
0, 0, 9, 0, 0, 10, 0, 1, 0, 0, 0, 0, 0, 4, 0, 0,
0, 0, 0, 0, 13, 6, 0, 0, 0, 14, 0, 0, 15, 12, 0, 16,
6, 5, 10, 0, 8, 2, 0, 0, 0, 12, 0, 0, 0, 1, 0, 7,
9, 0, 5, 4, 1, 0, 0, 2, 0, 0, 0, 0, 12, 0, 7, 0,
0, 0, 0, 0, 11, 0, 0, 13, 0, 3, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 16, 0, 0, 0, 13, 10, 15, 9, 14, 0, 4, 0,
10, 0, 0, 11, 0, 4, 8, 15, 0, 0, 0, 0, 5, 0, 13, 0,
0, 11, 0, 1, 0, 0, 0, 0, 10, 7, 4, 0, 3, 0, 0, 6,
0, 7, 0, 2, 14, 16, 6, 10, 0, 0, 0, 11, 0, 0, 0, 0,
16, 0, 0, 0, 0, 0, 1, 0, 12, 0, 0, 14, 0, 0, 0, 0,
0, 4, 0, 10, 0, 0, 0, 0, 15, 0, 0, 2, 16, 5, 0, 11,
11, 0, 12, 0, 0, 0, 14, 0, 0, 0, 13, 7, 0, 9, 6, 2,
8, 0, 7, 9, 0, 0, 11, 0, 0, 0, 14, 10, 0, 0, 0, 0,
```

```
0, 0, 4, 0, 0, 0, 0, 0, 11, 0, 2, 0, 0, 8, 0, 0,
0, 6, 0, 0, 12, 0, 0, 0, 9, 8, 0, 0, 0, 14, 1, 0]
```

E obtemos como resultado:

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

- N=5

Passamos à função *sudokuSolver* os parâmentros n=5 e o seguinte *sudoku*:

```
init5 = [0, 0, 12, 6, 0, 0, 7, 0, 18, 0, 5, 24, 0, 10, 1, 0, 0, 4, 0,
    0, 0, 0, 0, 0, 0,
2, 0, 19, 0, 13, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 18, 5, 0, 0, 0,
    0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 22, 0, 0, 0, 0, 3, 0, 2, 0, 0, 14, 12, 0, 16, 8,
    25, 0, 0,
0, 16, 0, 0, 0, 2, 23, 0, 0, 13, 12, 22, 0, 0, 0, 21, 15, 19, 3, 0, 0,
    0, 0, 14, 0,
23, 0, 24, 0, 0, 0, 0, 0, 25, 8, 4, 0, 16, 19, 21, 0, 0, 7, 0, 0, 0,
    3, 12, 0, 9,
0, 4, 0, 2, 0, 0, 0, 0, 0, 0, 0, 10, 0, 24, 12, 17, 16, 0, 0, 0, 5, 0,
    0, 0, 0,
0, 0, 9, 0, 0, 6, 25, 0, 0, 0, 8, 0, 5, 3, 0, 0, 0, 0, 0, 0, 20, 0, 0,
    18, 19,
15, 0, 10, 11, 0, 0, 0, 18, 12, 19, 0, 0, 0, 0, 0, 0, 0, 23, 0, 0, 7,
    0, 0, 4, 0,
0, 0, 0, 0, 0, 0, 0, 14, 0, 22, 0, 0, 18, 16, 20, 0, 6, 11, 13, 0, 0,
    0, 0, 0, 0,
0, 22, 0, 25, 0, 0, 1, 17, 5, 4, 7, 0, 0, 14, 0, 8, 3, 21, 0, 0, 11,
    0, 0, 0, 6,
0, 20, 13, 15, 0, 0, 0, 0, 0, 0, 9, 0, 0, 2, 0, 25, 0, 1, 8, 0, 0, 5,
    0, 21, 0,
0, 1, 0, 0, 0, 0, 16, 10, 0, 7, 0, 0, 4, 20, 0, 0, 9, 0, 0, 14, 0, 24,
    0, 17, 0,
25, 2, 5, 0, 0, 0, 0, 0, 13, 0, 0, 0, 0, 0, 22, 0, 0, 0, 0, 0, 19, 1,
    8, 0, 0,
0, 0, 7, 21, 0, 0, 12, 0, 2, 17, 0, 0, 0, 18, 6, 16, 0, 0, 15, 0, 0,
    13, 0, 10, 0,
```

```
8, 10, 18, 12, 16, 9, 0, 0, 0, 5, 0, 0, 0, 0, 19, 0, 0, 17, 0, 21, 0,
    15, 0, 0, 22,
0, 8, 0, 0, 15, 0, 3, 0, 6, 0, 21, 0, 0, 7, 0, 18, 14, 5, 0, 1, 0, 0,
    0, 0, 0,
0, 0, 0, 19, 0, 1, 0, 16, 11, 0, 0, 0, 10, 22, 25, 15, 0, 0, 0, 0, 0,
    0, 21, 0, 0,
0, 3, 1, 0, 21, 0, 0, 4, 0, 0, 0, 0, 2, 0, 13, 0, 24, 25, 0, 0, 14, 0,
    0, 6, 0,
0, 0, 0, 0, 0, 0, 0, 15, 0, 12, 14, 0, 6, 17, 24, 0, 0, 0, 0, 0, 0, 0,
    13, 0, 0,
0, 5, 23, 16, 4, 0, 13, 24, 7, 2, 0, 9, 0, 0, 15, 3, 0, 22, 0, 0, 0,
    0, 0, 0, 8,
0, 0, 25, 20, 2, 0, 19, 0, 0, 0, 0, 1, 0, 0, 0, 0, 21, 3, 0, 0, 12, 0,
    0, 0, 0,
16, 12, 0, 5, 0, 11, 21, 0, 23, 0, 0, 15, 0, 0, 0, 0, 19, 9, 0, 0, 0,
    0, 0, 25, 10,
0, 0, 0, 0, 9, 20, 22, 7, 4, 0, 3, 0, 14, 25, 18, 0, 11, 0, 0, 0, 0,
    0, 1, 0, 15,
24, 0, 6, 0, 22, 8, 0, 25, 14, 0, 10, 11, 0, 9, 0, 20, 1, 16, 0, 7, 0,
    23, 0, 0, 13,
14, 13, 21, 1, 0, 0, 5, 0, 0, 0, 6, 0, 22, 0, 23, 10, 0, 0, 0, 2, 0,
    0, 18, 7, 11]
```

E obtemos como resultado:

```
3 14 12 6 25 19 7 21 18 16 5 24 9 10 1 13 23 4 20 8 22 11 17 15 2
2 9 19 8 13 12 20 3 10 11 17 7 23 15 14 22 25 18 5 16 4 21 6 24 1
21 18 15 7 5 4 6 22 17 1 13 20 3 11 2 24 10 14 12 9 16 8 25 19 23
1 16 11 4 20 2 23 9 24 13 12 22 25 6 8 21 15 19 3 17 10 7 5 14 18
23 17 24 22 10 15 14 5 25 8 4 18 16 19 21 6 2 7 1 11 13 3 12 20 9
18 4 3 2 6 13 11 8 20 23 19 10 21 24 12 17 16 15 7 25 5 9 22 1 14
13 7 9 14 23 6 25 2 15 21 8 4 5 3 11 1 12 10 22 24 20 17 16 18 19
15 21 10 11 8 16 9 18 12 19 22 6 13 1 17 2 5 23 14 20 7 25 3 4 24
5 19 17 24 1 7 10 14 3 22 23 25 18 16 20 9 6 11 13 4 15 12 2 8 21
20 22 16 25 12 24 1 17 5 4 7 2 15 14 9 8 3 21 19 18 11 10 23 13 6
4 20 13 15 17 3 24 23 22 14 9 12 11 2 10 25 18 1 8 19 6 5 7 21 16
6 1 22 23 19 21 16 10 8 7 15 13 4 20 3 5 9 12 2 14 18 24 11 17 25
25 2 5 3 14 18 15 11 13 20 16 17 24 21 22 4 7 6 10 23 19 1 8 9 12
9 24 7 21 11 25 12 19 2 17 1 5 8 18 6 16 22 20 15 3 23 13 14 10 4
8 10 18 12 16 9 4 6 1 5 25 14 7 23 19 11 13 17 24 21 3 15 20 2 22
10 8 2 13 15 22 3 20 6 25 21 23 19 7 4 18 14 5 11 1 24 16 9 12 17
12 6 14 19 24 1 18 16 11 9 20 3 10 22 25 15 17 8 23 13 2 4 21 5 7
22 3 1 9 21 23 17 4 19 10 11 8 2 5 13 7 24 25 16 12 14 18 15 6 20
11 25 20 18 7 5 8 15 21 12 14 16 6 17 24 19 4 2 9 10 1 22 13 23 3
17 5 23 16 4 14 13 24 7 2 18 9 1 12 15 3 20 22 21 6 25 19 10 11 8
7 11 25 20 2 10 19 13 9 6 24 1 17 8 16 23 21 3 18 15 12 14 4 22 5
16 12 4 5 18 11 21 1 23 3 2 15 20 13 7 14 19 9 17 22 8 6 24 25 10
19 23 8 10 9 20 22 7 4 24 3 21 14 25 18 12 11 13 6 5 17 2 1 16 15
24 15 6 17 22 8 2 25 14 18 10 11 12 9 5 20 1 16 4 7 21 23 19 3 13
14 13 21 1 3 17 5 12 16 15 6 19 22 4 23 10 8 24 25 2 9 20 18 7 11
```

- N=6

Não foi possível obter resultados para N=6, pois o nosso código não estava suficientemente otimizado para que as máquinas em que foi testado o pudessem executar em tempo útil, não possuindo assim poder computacional suficiente. Tentamos otimizar, recorrendo ao *np.packbits()*, no entanto só funciona para listas de listas e o nosso exemplo de *sudoku* é uma lista.

3 Sistema de tráfego

3.1 Descrição do problema

Um sistema de tráfego é representado por um grafo orientado e ligado em que os arcos denotam vias de comunicação (que podem ter um ou dois sentidos de comunicação), e nodos denotam pontos de acesso.

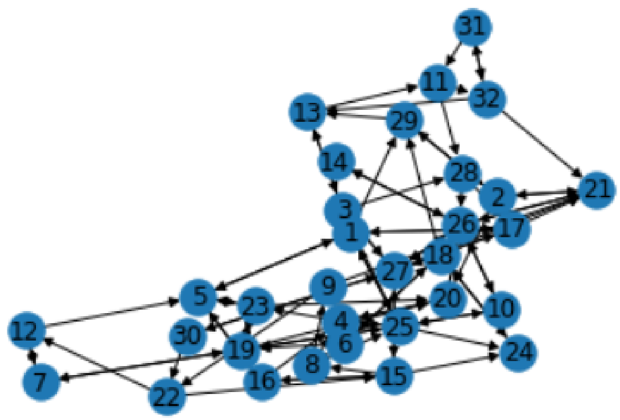

Figura 2: Exemplo de sistema de tráfego com 32 nodos

O exemplo de grafo apresentado anteriormente representa um sistema de tráfego, onde cada arco retrata uma via de comunicação e cada um dos trinta e dois nodos denota um ponto de acesso. Para além disso, o grafo tem de ser ligado, isto é, entre cada par de nodos $\langle n_1, n_2 \rangle$ tem de existir um caminho entre n1 e n2 e um caminho entre n2 e n1.

Por sua vez, o problema consiste em gerar um grafo ligado e posteriormente determinar o maior número de vias que é possível remover, mantendo o grafo ligado. Neste sentido, iremos dividir o problema em dois pontos:

1. Produzir um grafo ligado

Com recurso ao *NetworkX* iremos gerar aleatoriamente um grafo ligado com 32 nodos, considerando que existe igual probabilidade de cada via de comunicação ter um só sentido ou os dois sentidos. Como o grafo é ligado, para cada nodo, terá de existir pelos menos um ramo descendente. De acordo com o enunciado do trabalho, iremos assumir a probabilidade $p = 2^{-k}$ para a existência de k ramos descendentes adicionais.

2. Interromper vias de comunicação

De acordo com o enunciado do trabalho, pretende-se fazer manutenção ao sistema de tráfego e consequentemente determinar o maior número de vias que é possível remover mantendo mantendo o grafo ligado.

3.2 Concepção

De acordo com o que aprendemos nas aulas teóricas desta unidade curricular é possível concluir que este problema é um problema de Corte Máximo, ou seja, um problema de *bin packing* (ou problema do empacotamento). Assim sendo, teremos os seguintes objetivos e restrições:

3.2.1 Objetivo

$$maximize \quad (\sum_i x_i)$$

3.2.2 Restrições

A restrição para este problema é a seguinte:

$$\forall_{j \in N}. \quad (\sum_{i < linhas - 1} x_i * A_{i,j}) < 1$$

- N é o número de nodos do grafo
- x_i é uma variável binária
- A é a matriz de incidência

3.3 Resolução do problema

3.3.1 Algoritmos, testes realizados e resultados obtidos

Inicialmente, importamos as bibliotecas que nos serão úteis:

```python
import random
import math
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from pyscipopt import Model, quicksum
```

1.Produzir um grafo ligado

Para resolver o primeiro ponto, duas funções: a *criaGrafo* e a *verificaGrafo*. A função *criaGrafo* recebe como parâmetro o número de nodos e cria um grafo obdecendo aos requisitos e probabilidades referidas no enunciado.

```
N=32

def criaGrafo(N):


    #criar um grafo vazio
    grafo = nx.DiGraph()

    #criar nodos
    grafo.add_nodes_from(range(1,N+1))


    for i in grafo.nodes:

        #1) cada nodo ter de existir pelos menos um ramo descendente:
        t=np.random.choice(grafo.nodes)
        opt=random.choice([1,2])
        if(opt>1):
            grafo.add_edge(i,t)
            grafo.add_edge(t,i)
        else:
            grafo.add_edge(i,t)


        #2) Ramos descendentes adicionais
        for j in range(1, N-1):
            #prob2=(math.pow(2,-j))/2
            prob=(math.pow(2,-j))
            opt=random.choice([1,2])
            e=np.random.choice(grafo.nodes)

            if (random.random() < prob):

                if (opt<2):
                    grafo.add_edge(i,e)
                else:
                    grafo.add_edge(i,e)
                    grafo.add_edge(e,i)

    return(grafo)
```

Por sua vez a função *verificaLigado* irá assegurar que entre cada par de nodos $\langle n_1, n_2 \rangle$ existe um caminho entre n1 e n2 e um caminho entre n2 e n1. Optamos por utilizar esta função para assegurar a existência de caminhos, ao invés de adicionar ramos, de modo a garantir o cumprimento das probabilidades. Caso fossem adicionados vértices as probabilidades seriam adulteradas.

```
def verificaLigado(graph):
    flag = 1

```

```
    #print(nx.is_connected(graph))

    while(flag==1):

        if nx.is_strongly_connected(graph):
            flag=0
        else:
            graph=criaGrafo(N)

    return graph
```

Assim sendo, após a execução de:

```
graph=criaGrafo(N)
grafo=verificaLigado(graph)
nx.draw_networkx(grafo)
```

Obtemos como resultado:

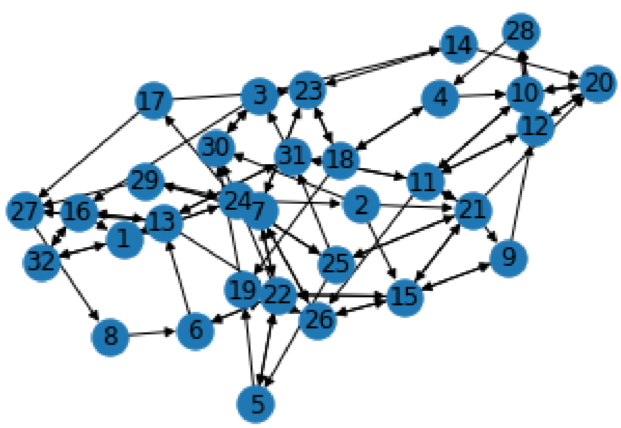

Figura 3: Exemplo de grafo ligado com 32 nodos

2. Determinar quantas vias de comunicação é possível interromper

Com o objetivo de determinar quantas vias de comunicação é possível interromper de modo a proceder à manutenção do sistema. Criamos as funções *determinaCaminhos* que irá determinar quais os caminhos que existem no grafo e a função *CorteMaxCaminhos* que iremos utilizar para efetuar o corte máximo do grafo, para além disso recorremos a *nx.adjacency_matrix(grafo)).toarray()* para determinar a matriz de adjacência do grafo.

```python
def determinaCaminhos(grafo, N):

    caminhos = []
    for i in range(1,N+1):
        for j in range(i+1,N+1):
                caminhos.append(nx.all_simple_paths(grafo, i, j))
                caminhos.append(nx.all_simple_paths(grafo, j, i))

    return caminhos

caminhos = determinaCaminhos(grafo,N)
```

```python
matrizI = (nx.adjacency_matrix(grafo)).toarray()
#print(matrizI)
```

```python
def CorteMaxCaminhos(A):
    x={}
    cover = Model()


    linhas = len(A)
    colunas = len(A[0])

    for i in range(linhas):
        x[i] = cover.addVar(str(i), vtype = 'B')

    for j in range(colunas):
        cover.addCons(quicksum([(x[i] * A[j][i]) \
                          for i in range(linhas)]) <= 1)

    cover.setObjective(quicksum(x.values()), sense = 'maximize')
    cover.optimize()

    if cover.getStatus() == 'optimal':
        return[(i) for i in range(colunas) if cover.getVal(x[i]) == 1]

print( "  possvel  interromper" , len(CorteMaxCaminhos(matrizI)),
    "vias de comunicao.")
```

E obtemos como resposta:

```
É possível interromper 12 vias de comunicação.
```

Figura 4: Resultado da função CorteMaxCaminhos para o grafo anterior