tp1exc2

October 1, 2023

1 Sistema de tráfego

Grupo 05

Eduardo André Silva Cunha
Gonçalo Emanuel Ferreira Magalhães A100084

2 Problema

1. Um sistema de tráfego é representado por um grafo orientado ligado. Os nodos denotam pontos de acesso e os arcos denotam vias de comunicação só com um sentido . O grafo tem de ser ligado: entre cada par de nodos $\langle n_1, n_2 \rangle$ tem de existir um caminho $n_1 \rightsquigarrow n_2$ e um caminho $n_2 \rightsquigarrow n_1$.
    1. Gerar aleatoriamente o grafo com $N \in \{8..15\}$ nodos e com ramos verificando:
        1. Cada nodo tem um número aleatório de descendentes $d \in \{0..3\}$ cujos destinos são também gerados aleatoriamente.
        2. Se existirem "loops" ou destinos repetidos, deve-se gerar outro grafo.
    2. Pretende-se fazer manutenção interrompendo determinadas vias. Determinar o maior número de vias que é possível remover mantendo o grafo ligado.

2.0.1 Import

Imports usados

[1]:
```
from ortools.linear_solver import pywraplp
import networkx as nx
import random
import copy
```

2.0.2 Criação do grafo:

Inicialmente estavamos a trabalhar apenas com grafos sob a forma de dicionário mas após a aula prática de quarta feira decidimos alterar isso.

[2]:
```
def cria_grafo():
    while True:
        nr_nodos = random.randint(8, 15)
        grafo_dict = {}
```

```
        # Inicializar grafo como um dicionário de listas
        for i in range(0, nr_nodos):
            grafo_dict[i] = []

        # Criar grafo aleatório
        for i in range(0, nr_nodos-1):
            nr_saidas = random.randint(0, 3)
            for j in range(nr_saidas):
                destino = random.randint(0, nr_nodos-1)
                if destino != i:
                    grafo_dict[i].append(destino)

        grafo_nx = nx.Graph(grafo_dict) # Converter o grafo de dicionário de listas para NetworkX

        if nx.is_connected(grafo_nx) and grafo_valido(grafo_nx): # Verifica se o grafo é conectado e valido conforme o enunciado
            return grafo_nx

def grafo_valido(grafo):
    num_nodes = len(grafo)
    if num_nodes < 8 or num_nodes > 15: # Verifica o número de nós (8 a 15 nodos)
        return False

    for node in grafo.nodes():
        if len(list(grafo.neighbors(node))) > 3: # Verifica se cada nó tem no máximo 3 destinos
            return False

    for edge in grafo.edges(): # Verifica se não há loops (arestas de um nó para ele mesmo)
        if edge[0] == edge[1]:
            return False

    # Se todas as verificações passaram, o grafo é válido
    return True


grafo = cria_grafo()
# Impressão do grafo inicial:
grafo_inicial = nx.to_dict_of_lists(grafo)
print("Grafo gerado:")
print(grafo_inicial)
```

```
Grafo gerado:
{0: [8, 1], 1: [0, 9], 2: [9, 5, 6], 3: [9, 7], 4: [8], 5: [2], 6: [7, 2, 10],
```

```
7: [6, 3, 10], 8: [0, 4], 9: [1, 2, 3], 10: [6, 7]}
```

2.0.3 Resolução do problema

Inicialmente tentamos resolver este problema com o uso de um solver "SCIP" criando uma variavel X que a cada par de arestas correspondia um valor 0 ou 1 que representava se a aresta seria removida ou não, respetivamente.

[3]:
```
# Inicialização do solver:
solver = pywraplp.Solver.CreateSolver('SCIP')

# Variáveis de decisão: x[a] == 1 se a aresta 'a' for removida, 0 caso␣
 ↪contrário.
x = {}
for edge in grafo_inicial.keys():
    for neighbor in grafo_inicial[edge]:
        if (edge, neighbor) in grafo.edges:
            x[(edge, neighbor)] = solver.IntVar(0, 1, f'x[{edge},{neighbor}]')
```

2.0.4 Restrição

1. Manter o grafo conectado Potencialmente incorreto, a nossa ideia era garantir que entre cada 2 nodos com ligação inical essa ligação mantinha-se, mas não se verifica...

[4]:
```
for node in grafo_inicial.keys():
    for neighbor in grafo_inicial[node]:
        if (node, neighbor) in x:
            solver.Add(x[(node, neighbor)] + x[(neighbor, node)] >= 1)
```

2.0.5 Objetivo

1. Maximizar o numero de arestas removidas, ou seja neste caso, maximizar o numero de valores iguais a 1 na variavel "X"

[5]:
```
objective = solver.Objective()
for edge in grafo_inicial.keys():
    for neighbor in grafo_inicial[edge]:
        if (edge, neighbor) in x:
            objective.SetCoefficient(x[(edge, neighbor)], 1)  # Coeficiente 1␣
 ↪para todas as variáveis x

objective.SetMaximization()
```

2.0.6 Impressão do grafo final

[6]:
```
solver.Solve()

# Arestas removidas
removed_edges = [(edge[0], edge[1]) for edge in x if x[edge].solution_value()␣
 ↪== 1]

# Criar o novo grafo resultante após a remoção das arestas
grafo_resultante = grafo.copy()
grafo_resultante.remove_edges_from(removed_edges)

# Imprimir o grafo resultante como dicionário de listas
dict_of_lists = nx.to_dict_of_lists(grafo_resultante)
print("Grafo resultante como dicionário de listas:")
print(dict_of_lists)
```

```
Grafo resultante como dicionário de listas:
{0: [], 1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: [], 9: [], 10: []}
```

3 Nova Resolução:

Através da utilização do Solver "SCIP", não conseguimos implementar uma restrição que garantisse que o grafo final permanecesse conectado, então após a aula de quarta-feira(27/9), onde nos foi apresentado a biblioteca "nx" decidimos resolver o problema utilizando funções da biblioteca "nx".

Todo o código de criação do grafo foi mantido.

[7]:
```
grafo = cria_grafo()
grafo_inicial = nx.to_dict_of_lists(grafo)
print("Grafo gerado:")
print(grafo_inicial)
```

```
Grafo gerado:
{0: [5, 6], 1: [6], 2: [3, 4], 3: [2, 5], 4: [2], 5: [0, 3, 7], 6: [1, 0], 7:
[5]}
```

Usamos a função pré-definida para encontrar a árvore geradora mínima.

[8]:
```
minimum_spanning_tree = nx.minimum_spanning_tree(grafo)
```

Guardamos em arestas_removidas todas as arestas que não são necessárias para a árvore geradora mínima.

[9]:
```
arestas_removidas = [edge for edge in grafo.edges() if edge not in␣
 ↪minimum_spanning_tree.edges()]
```

Remoção das arestas desnecessárias no grafo

[10]:
```
grafo_modificado = grafo.copy()
grafo_modificado.remove_edges_from(arestas_removidas)
```

Conversão do grafo modificado de volta para o formato de dicionária de listas.

[11]:
```
grafo_modificado_dict = nx.to_dict_of_lists(grafo_modificado)
```

3.0.1 Prints dos resultados obtidos

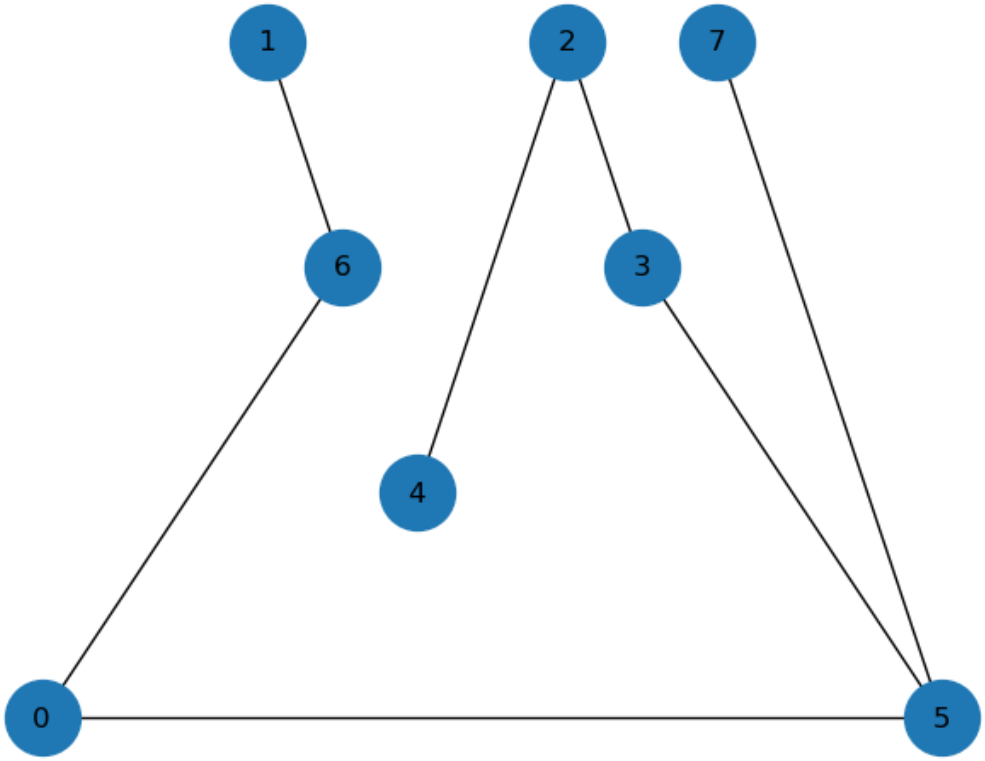
[12]:
```
print("Arestas Removidas:")
print(arestas_removidas)

print("Grafo modificado:")
print(grafo_modificado_dict)
```

```
Arestas Removidas:
[]
Grafo modificado:
{0: [5, 6], 1: [6], 2: [3, 4], 3: [2, 5], 4: [2], 5: [0, 3, 7], 6: [0, 1], 7:
[5]}
```

3.0.2 Impressão do desenho do grafo

[13]:
```
pos = nx.planar_layout(grafo_modificado)
nx.draw(grafo_modificado, pos, with_labels=True,node_size=1000)
```


1
2
7
6
3
4
0
5