Universidade Federal de São Carlos - UFSCar

João Vitor Azevedo Marciano - 743554

Trabalhos práticos de Teoria dos Grafos

São Carlos - SP 2018

Os trabalhos tem como objetivo dar ao aluno uma noção mais prática da teoria dos grafos e sua importâncias em diversos contextos de problemas do mundo real.

Os 5 escolhidos entre os propostos foram:

- **PROJETO 2: ÁRVORE GERADORA MÍNIMA**
- **PROJETO 3: BUSCA EM LARGURA E PROFUNDIDADE**
- **Projeto 7: Sudoku Problem Solver**
- **PROJETO 8: EMPARELHAMENTOS ESTÁVEIS E O ALGORITMO DE GALE-SHAPPLEY**
-
- **PROJETO 10: FLUXO MÁXIMO E O ALGORITMO DE FORD-FULKERSON**

Descrição da execução do experimento

PROJETO 2: ÁRVORE GERADORA MÍNIMA

A partir de um dataset específico (grafo ponderado armazenado em arquivo .gml, .graphml, .txt, .net, etc) implementar o algoritmo de Kruskal ou de Prim para extrair uma Minimum Spanning Tree (MST) de G.
LINK: Grafo HA30 com 30 cidades do mundo

A MST(ou Minimum Spanning Tree) é muito relevante em diversas áreas da computação, como projetos de redes, aproximações para problemas NP-Hard(ex: TSP), reduzir o tamanho de dados armazenados quando sequenciando aminoácidos em uma proteína, k-clustering, etc.
Existem dois algoritmos que resolvem o problema de maneira ótima: o algoritmo de Kruskal, que faz crescer uma floresta geradora até que ela se torne uma árvore; e o algoritmo de Prim, que faz crescer uma árvore mínima, até que ela se torne geradora.
Foi escolhido o algoritmo de Prim como solução para esse projeto, implementado a seguir na linguagem Python.
"""
-Created on Fri Dec 7 08:09:52 2018

https://networkx.github.io/documentation/networkx-1.10/reference/introduction.html

@author: MartManHunter
"""

```
import networkx as nx
import numpy as np
from matplotlib import pyplot as plt

def Prim(g, s):
    minCin = {}               ##Menor weight de entrada para v até agora
    predecessor = {}          ##predecessor de v na arvore ( vertice pelo qual entrei no v atual)
    for v in g.nodes():       #ini  de quem não é inicial -> sem predecessor e distancia infinita da ""raiz""
            minCin[v] = np.inf
            predecessor[v] = 'null'
  #Inicialização do nó inicial ""RAIZ""
    minCin[s] = 0
    MST = nx.create_empty_copy(g)
```

```
    while minCin:
            u = min(minCin,key = minCin.get)
            del minCin[u]
            for vizinho in g[u]:
                    if vizinho in minCin and g[u][vizinho]['weight'] < minCin[vizinho]:
                            predecessor[vizinho] = u
                            minCin[vizinho] = g[u][vizinho]['weight']
            if predecessor[u] is not 'null':
                    for v1,v2,data in g.edges(data=True):
                            if (v1 == predecessor[u] and v2 == u):          ##se v1 é predecessor de u
                                    MST.add_edge(v1,v2, weight=data['weight'])   #add ## (u,v) na arvore
                            elif (v1 == u and v2 == predecessor[u]):
                                    MST.add_edge(v2,v1, weight=data['weight'])
                                    MST_final = MST.copy()
    return MST_final

A = np.loadtxt('ha30_dist.txt')         ##trecho de código
g = nx.from_numpy_matrix(A)             ##fornecido pelo professor, só aceitei

tree = Prim(g, 0)
nx.draw_networkx(tree)
plt.show()
```

O algoritmo aproveita a propriedade do corte, que diz que se existe um corte(A,B), a aresta de custo mínimo que atravessa o corte faz parte da MST. Utilizando-se deste teorema, o algoritmo inicializa o corte como (s, V - {s}) e segue ampliando-o a cada iteração, fazendo escolhas gulosas esperando que elas levem ao resultado ótimo.

Projeto 7: Sudoku Problem Solver com Coloração de Vértices

O programa deve, a partir de um tabuleiro de SUDOKU, criar um grafo de incompatibilidade através das regras do jogo. Considere a seguinte instância do jogo:

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

A regra básica do jogo é: preencha os números de forma que não haja repetição na mesma linha, coluna ou quadradinho (3 x 3).

METODOLOGIA

Em outras palavras, o modelo a ser gerado é um grafo G com 81 vértices (quadrado 9 x 9), um para cada casa do tabuleiro. Cada célula do tabuleiro deve ser nomeada como um vértice, percorrendo o tabuleiro linha a linha (v1, v2, v3,..., v81). A partir do grafo nulo inicial (apenas vértices), arestas devem ser adicionadas da seguinte forma:

i) Se um vértice vi pertence a mesma linha que um vértice diferente vj então a aresta (vi, vj) deve ser adicionada a G

ii) Se um vértice vi pertence a mesma coluna que um vértice diferente vj então a aresta (vi, vj) deve ser adicionada a G
iii) Se um vértice vi pertence ao mesmo quadrado (3 x 3) que um vértice diferente vj então a aresta (vi, vj) deve ser adicionada a G

Logo após a geração do grafo, a pré-coloração inicial (dada pelos quadradinhos já preenchidos do tabuleiro) deve ser realizada. Por exemplo, no caso ilustrado pela figura acima, as cores dos vértices da primeira linha serão: v1 = 9, v2 = 5, v6 = 1, v9 = 8. De posse do grafo e da pré-coloração inicial, o objetivo do trabalho consiste em desenvolver um solucionador automático para esse jogo utilizando o algoritmo para coloração de vértices conhecido como Welsh & Powell. Note que a pré-coloração faz com que as listas iniciais de possíveis cores dos vértices do grafo sejam diferentes. Na prática, para um vértice vi, devemos inicialmente remover de sua lista todas as cores encontradas em seus vizinhos. A heurística a ser adotada consiste em começar a colorir os vértices cujas listas de cores são as menores possíveis, pois assim praticamente não há dúvidas sobre a cor que devem receber. Por exemplo, se inicialmente um vértice possui uma lista de cores com apenas 1 cor, devemos iniciar por ele pois só há obrigatoriamente uma opção válida. Se em algum momento a lista de um vértice se esvazia ou tem apenas cores já assumidas pelos vizinhos, então não há solução válida por esse caminho e possivelmente para o tabuleiro em questão. Em geral, nesses casos é necessário aumentar a pré-coloração inicial, isto é, preencher mais quadrados inicialmente.

Funções que resolvem o problema do Sudoku:

```
import networkx as nx
import sys

nodes = [['00','01','02','03','04','05','06','07','08'],
        ['10','11','12','13','14','15','16','17','18'],
        ['20','21','22','23','24','25','26','27','28'],
        ['30','31','32','33','34','35','36','37','38'],
        ['40','41','42','43','44','45','46','47','48'],
        ['50','51','52','53','54','55','56','57','58'],
        ['60','61','62','63','64','65','66','67','68'],
        ['70','71','72','73','74','75','76','77','78'],
        ['80','81','82','83','84','85','86','87','88']]

quadradin = [ ['00','01','02','10','11','12','20','21','22'],
            ['03','04','05','13','14','15','23','24','25'],
            ['06','07','08','16','17','18','26','27','28'],
            ['30','31','32','40','41','42','50','51','52'],
```

```
            ['33','34','35','43','44','45','53','54','55'],
            ['36','37','38','46','47','48','56','57','58'],
            ['60','61','62','70','71','72','80','81','82'],
            ['63','64','65','73','74','75','83','84','85'],
            ['66','67','68','76','77','78','86','87','88']]
## usado Welsh-Powel para pintar os vertices,
def welshpowell(g):
    for node in g.node:
        if not g.node[node]['status']:
            for e in g.neighbors(node):
                if g.node[e]['status']:
                    try:
                        g.node[node]['color'].remove(g.node[e]['color'])
                    except:
                        pass
 def update(g): ## verifica se um vertice possui apenasv uma cor em sua lista de cores.
#Se sim, pinta ele e marca como pintado. Caso contrário ele não faz nada.
    for node in g.node:
        if not g.node[node]['status'] and len(g.node[node]['color']) == 1:
            g.node[node]['status'] = True
            g.node[node]['color'] = g.node[node]['color'][0]


def clear(g):   #A função clear apenas limpa a lista dos vertices não necessários.
    for node in g.node:
        if g.node[node]['status'] and type(g.node[node]['color']) != int:
            g.node[node]['status'] = False


def engage(g): #A função engage considera v = True como e tenta retirar dos vizinhos
#uma lista. Então roda powell & update para receber o resultado mais “limpo” desse
#processo.
    for i in range(9):
        for j in range(9):
            if not g.node[nodes[i][j]]['status']:
                g.node[nodes[i][j]]['status'] = True
                welshpowell(g)
                update(g)
    clear(g)


def main(argv):
```

```
    try:
        nomeArquivo = 'teste.txt'##sua matriz com valores sep. por espaço aqui. x = vazio
    except IndexError:
        print ('deu tudo errado :/ ')
        sys.exit(-1)


    g = nx.Graph()

    # Arquivo que contém a matriz
    fp = open(nomeArquivo, 'r')
    data = fp.read().split('\n')
    data.remove('')
    sudoku = []
    for line in data:
        sudoku.append(line.split(' '))

    # criando vértices com cores e status
    for i in range(9):
        for j in range(9):
            if sudoku[i][j] == 'x':     ##ideia: colorir os sem valor primeiro, e ficarão menos
cores para colorir os que possuem valores
                g.add_node(nodes[i][j], color=[1,2,3,4,5,6,7,8,9], status=False)
            else:
                g.add_node(nodes[i][j], color=int(sudoku[i][j]), status=True)

    # na linha
    for i in range(9):
        for j in range(8):
            for k in range(j,8):
                g.add_edge(nodes[i][j], nodes[i][k+1])

    # na coluna
    for i in range(8):
        for j in range(9):
            for k in range(i,8):
                g.add_edge(nodes[i][j], nodes[k+1][j])

    #no quadradinho
    for i in range(9):
        for j in range(8):
            for k in range(j,8):
```

```
                g.add_edge(quadradin[i][j], square[i][k+1])

    for i in range(9):
        engage(g)
        welshpowell(g)
        update(g)


    for i in range(9):
        for j in range(9):
            print(g.node[nodes[i][j]]['color'])
            print()
    nx.draw_networkx(g)

    ##print não está o mais perfeito do mundo :/
    ##linhas é que são escritas, e não colunas

if __name__ == '__main__':
    main(sys.argv)
```

O algoritmo apresentou a resposta correta em todos os testes realizados.

In-1

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

Out 1

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

QUESTIONAMENTOS

a) O que podemos dizer sobre os graus do grafo G resultante?

Todos os graus são iguais, já que todo v se conecta com seus 8 vizinhos da linha, os 8 da coluna, e os 8 do seu ‘quadrado’.

b) Utilizando o tabuleiro acima, obtenha uma solução aplicando o algoritmo de Welsh & Powell.

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

c) Reaplique o algoritmo mais três vezes e compare as 4 soluções obtidas. São iguais ou diferentes?

d) Ainda considerando o tabuleiro acima, note que inicialmente existem 45 valores iniciais. Remova 2 valores quaisquer de cada linha (18 no total), totalizando 27 células preenchidas. Execute o método. Foi possível obter uma solução válida? Compare a solução com as obtidas anteriormente.

e) O que acontece se a pré-coloração inicial tiver apenas 9 valores (1 por linha)? Pense em estratégias/heurísticas que podem auxiliar no processo de coloração. Em http://rachacuca.com.br/logica/sudoku/ você pode encontrar algumas.

INDO ALÉM

Pesquise na internet sobre o jogo, mais precisamente, sobre o unicidade da solução em termos do número mínimo de elementos preenchidos iniciais. Em outras palavras, será que existe um número mínimo de quadradinhos que precisam ser preenchidos inicialmente para garantir que o jogo possui solução e ela seja a única possível?

Apenas dois algoritmos foram parcialmente desenvolvidos, devido à escassez de tempo/mãos.