Explorando a linguagem Python e a biblioteca NetworkX para a solução de problemas em grafos

Alisson Hayasi da Costa

Fernanda Malheiros Assi

Teoria dos Grafos – Departamento de Computação, Universidade Federal de São Carlos

https://github.com/a-hayasi/graphtheory2018

Grupo Desenvolvedor
Alisson Hayasi da Costa, 726494
Fernanda Malheiros Assi, 743534
Teoria dos Grafos – Turma A

Dezembro 2018

Credits
Template by Andrea Hidalgo on Overleaf
Photos by Diego PH on Unsplash

Contents

1 Introdução 5

1.1 Origem da Teoria dos Grafos 5

1.2 O que são Grafos? 5

1.3 Exemplos de Problemas Encontrados 5

1.4 Contextualização 6

1.5 Linguagem Python 6

1.6 NetworkX 6

2 Fundamentos Teóricos 7

2.1 Problema 1: Snakes and Ladders 7

2.1.1 Cadeias de Markov 7

2.1.2 Power Method 8

2.1.3 PageRank 9

2.1.4 Aplicação em Snakes and Ladders 9

2.2 Problema 2: Árvore Geradora Mínima 9

2.2.1 Algoritmo de Prim 10

2.3 Problema 3: Busca em Largura e Profundidade 11

2.3.1 Busca em Largura 11

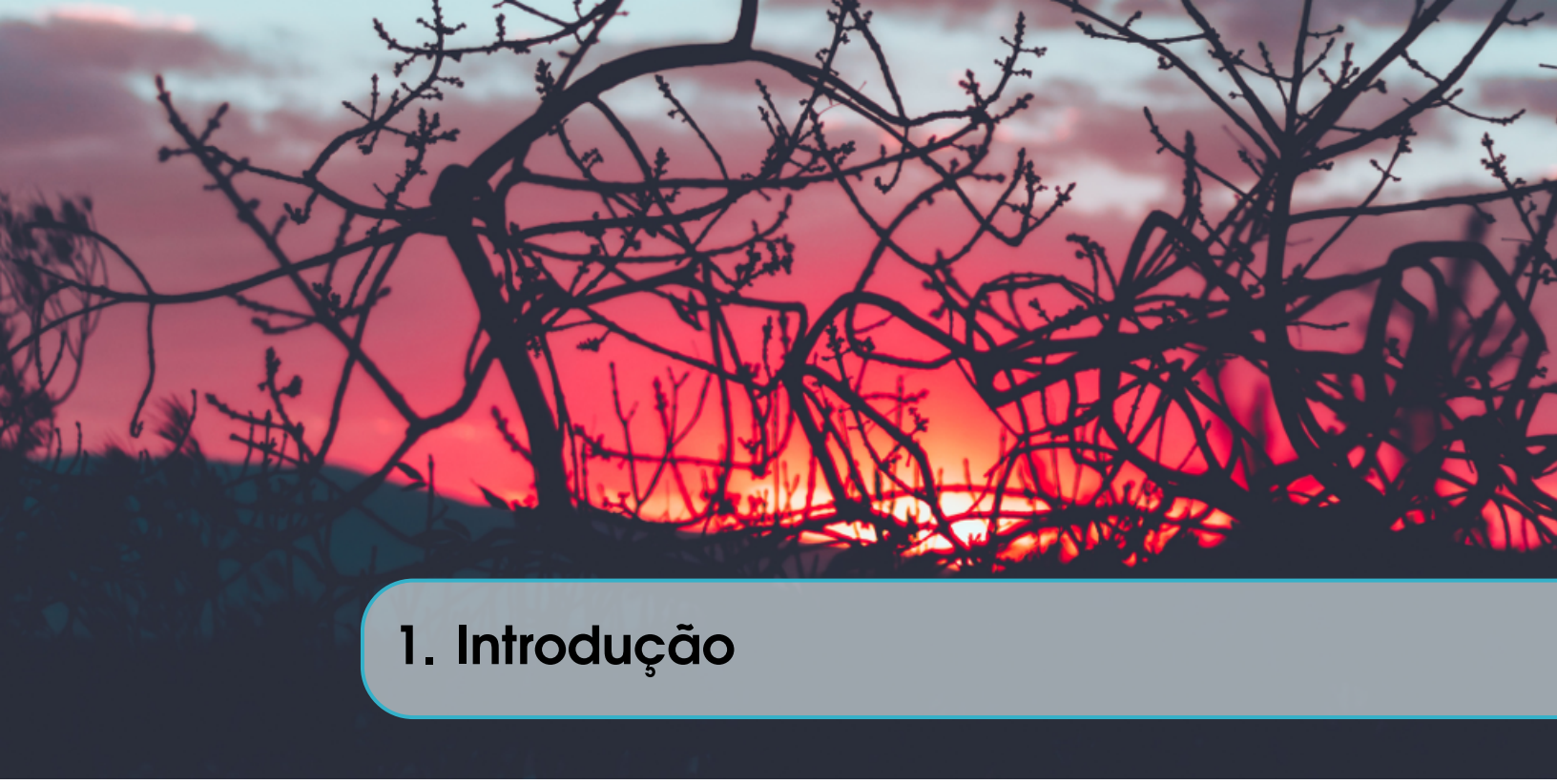
2.3.2 Busca em Profundidade 12

2.4 Problema 4: Árvores de Caminhos Mínimos e Agrupamento de Dados 14
2.4.1 O Algoritmo de Dijkstra 15
2.5 Problema 5: Sudoku Problem Solver Com Coloração De Vértices 16
2.5.1 Coloração de vértices 16

3 Materiais e Métodos 18
3.1 Bibliotecas e Pacotes 18
3.2 Ambiente de Desenvolvimento 18
3.3 Conjuntos de Dados 19

4 Resultados e Discussões 20
4.1 Solução 1: Snakes and Ladders 20
4.1.1 Repositórios e Códigos-fonte 24
4.2 Solução 2: Árvore Geradora Mínima 24
4.2.1 Repositórios e Códigos-fonte 27
4.3 Solução 3: Busca em Largura e Profundidade 27
4.3.1 Busca em Largura 27
4.3.2 Repositórios e Códigos-fonte 29
4.3.3 Busca em Profundidade 29
4.3.4 Repositórios e Códigos-fonte 31
4.4 Solução 4: Árvores de Caminhos Mínimos e Agrupamento de Dados 31
4.4.1 Repositórios e Códigos-fonte 33
4.5 Solução 5: Sudoku Problem Solver Com Coloração De Vértices 34
4.5.1 Questionamentos 36
4.5.2 Repositórios e Códigos-fonte 37

5 Considerações Finais 38



1. Introdução

1.1 Origem da Teoria dos Grafos

A teoria dos grafos teve origem em 1736, quando o alemão Leonhard Paul Euler (1707-1783) publicou um artigo sobre o problema das *Sete pontes de Königsberg*. Este problema consistia em tentar achar um caminho que passava por todas as 7 pontes que dividiam a cidade de Königsberg em 4.

Embora a primeira vista o problema pareça trivial, ele não possui solução. Para provar isso, Euler deu início ao desenvolvimento da Teoria dos Grafos e produzindo um novo ramo da matemática (topologia).

1.2 O que são Grafos?

Um grafo $G = (V, E)$ é um conjunto finito não vazio V de vértices e um conjunto E de arestas onde as arestas podem possuir um peso associado denotado por w.

Muitas vezes o grafo é utilizado como uma representação gráfica de um problema da vida real, para que assim a visualização e entendimento fique mais fácil. Normalmente, os pesos de um grafo indicam a distância ou custo para ir de um vértice a outro na vida real.

1.3 Exemplos de Problemas Encontrados

Há inúmeros problemas que podem ser aplicados na vida real utilizando a Teoria dos Grafos. Como já mencionado, o próprio estudo dos grafos começou a partir de um problema real. Outros exemplos de problemas são: encontrar caminhos mínimos em de um ponto a outro em uma cidade, alocar x tarefas para y pessoas, descobrir o fluxo máximo em uma rede, entre muitos outros.

1.4 Contextualização

A Teoria dos Grafos fornece uma grande quantidade de ferramentas e algoritmos para a solução de diversos tipos de problemas. Alguns problemas famosos envolvendo grafos são: Coloração de Grafos; Árvores de Caminhos Mínimos; Problemas de Roteamento e diversos outros mais complexos como o Caixeiro Viajante. Devido a elegância e eficiência dos métodos encontrados a partir da matemática discreta da Teoria dos Grafos, muitos problemas da vida real são solucionados através da aplicação de algum conceito gerado a partir da Teoria dos Grafos.

Este trabalho, portanto, busca elucidar conceitualmente o que é a Teoria dos Grafos além de abordar, conceitos, problemas e técnicas famosas e extremamente significantes para a resolução de problemas práticos vivenciados na vida real.

1.5 Linguagem Python

Python é uma das linguagens de programação mais utilizadas atualmente. Por ser de alto nível, seu código é fácil de ser interpretado. Além disso, ela possui uma gama de bibliotecas que implementam diversas funções, tornando a vida do programador muito mais fácil. Ela é uma linguagem *opensource*, ou seja, possui documentação aberta, o que torna o seu aprendizado muito mais acessível.

1.6 NetworkX

NetworkX é um biblioteca do Python utilizada especialmente criação, manipulação e estudo de grafos e redes complexas sejam essas estruturas estáticas ou dinâmicas. A biblioteca fornece uma grande quantidade de algoritmos para grafos já implementados de forma eficiente, além de fornecer diversas formas de criação de grafos,dígrafos e qualquer tipo de estrutura de redes e relações necessárias

2. Fundamentos Teóricos

2.1 Problema 1: Snakes and Ladders

Snakes and Ladders é um antigo jogo de tabuleiro originado no século II (d.C.) na Índia e jogado por dois ou mais jogadores. O objetivo do jogo é alcançar a última casa do tabuleiro sendo, o vencedor, aquele que chegar primeiro à última casa.

No entanto, durante a corrida no tabuleiro realizada através de dados, há casas com *cobras* e *escadas* que conectam dois quadrados específicos e promovem uma consequência específica. Isto é, caso o jogador se movimente para uma casa com uma cabeça de *cobra*, então, ele deve descer para a casa que se encontra no rabo da cobra; caso o jogador se movimente para uma casa do tabuleiro que possui o pé de uma escada, então,ele deve subir para a casa onde se encontra o fim da escada.

O jogo *Snakes and Ladders* atualmente possui um grande número de variações com diferentes dimensões do tabuleiro, número de *cobras*, *escadas* etc. Porém, o mais interessante é que qualquer variação de *Snakes and Ladders* pode ser representada como uma *Cadeia de Markov*. Essa representação permite responder questões como: Qual a chance do primeiro jogador vencer? Quantos movimentos são necessários para que algum jogador ganhe? Quais casas são mais prováveis de serem acessadas?

2.1.1 Cadeias de Markov

De forma geral, podemos descrever um processo estocástico como um fenômeno que varia em algum grau, de forma imprevisível, à medida que o tempo passa. Essa descrição nos permite compreender a evolução de um sistema ao longo do tempo e predizer a probabilidade de ocorrência de certos estados. Uma Cadeia de Markov é um processo estocástico caracterizado pelo fato de que o estado futuro depende apenas do estado atual e qualquer estado passado não tem nenhuma influência no estado futuro. Essa propriedade é denominada como Markoviana em homenagem ao matemático Andrei

Andreyevich Markov, e sua definição estabelece que a sequência de estados anteriores é indiferente pra se predizer o próximo estado.

Tanto os processos estocásticos quanto as Cadeias de Markov são muito importantes para diversas aplicações no mundo real, como, na Economia com o mercado de ações e na Física com a termodinâmica. Entretanto, uma das aplicações mais relevantes de Cadeias de Markov está na definição de um PageRank para uma página web na Internet. Já em jogos como *Snakes and Ladders*, podemos utilizar Cadeias de Markov para descrever a probabilidade de um jogador estar em um certo estado num determinado tempo.

Formalmente, podemos definir uma Cadeia de Markov homogênea de estados finitos e tempo discreto pela tupla.

$$CM = (S, x_k P, w^{(k)}) \text{ onde,} \tag{2.1}$$

- $S = \{s_1, s_2, ..., s_n\}$;
- x_k é uma variável aleatória que admite valores de S no tempo k;
- P é a matriz de probabilidades de transição de estados
- $w^{(k)}$ é o vetor de probabilidades de cada estado no tempo k.

Já a matriz P pode ser encontrada ao modelar o problema em questão como um grafo. Assim, seja $G = (V, E)$ um grafo básico simples com matriz de adjacência A. Podemos definir a matriz P como $P = \Delta^{-1}A$ onde,

$$\Delta^{(-1)} = \begin{bmatrix} \frac{1}{d(v_i)} & 0 & ... & 0 \\ 0 & \frac{1}{d(v_i)} & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & 0 & \frac{1}{d(v_i)} \end{bmatrix} \tag{2.2}$$

em que $\frac{1}{d(v_i)}$ é o inverso do grau do vértice i.

Com isso, temos a matriz de probabilidades de transição de estados e através disso podemos encontrar uma distribuição estacionária para o vetor $w^{(k)}$, ou seja, encontrar a probabilidade de se estar em cada vértice i (estado i) no tempo k. O método iterativo para se obter a distribuição estacionária em um tempo i é conhecido como *Power Method*.

2.1.2 Power Method

O *Power Method* é um método iterativo para se obter a distribuição de probabilidades dos estados em um tempo k. Isto é,

$$\overrightarrow{w}^{(k)} = \overrightarrow{w}^{(k-1)}P = (\overrightarrow{w}^{(k-2)}P)P = ((\overrightarrow{w}^{(k-3)}P)P)P = ... \tag{2.3}$$

logo,

$$\overrightarrow{w}^{(k)} = \overrightarrow{w}^{(0)} P^k \tag{2.4}$$

2.1.3 PageRank

Apesar da modelagem padrão da Cadeia de Markov solucionar uma quantidade relevante de problemas, isto não ocorre quando se trata de dígrafos pois a presença de *dangling nodes* produz vértices sem saída que tende a ocasionar em um estado imobilizador. Além disso, pode não existir k de forma que seja possível estar em qualquer estado com probabilidade não nula.

Portanto, para solucionar tais problemas, permitimos eventuais saltos com pequena probabilidade de ocorrência e *loops*. Formalmente, isso significa que a matriz P, inicialmente definida como P como $P = \Delta^{-1}A$, passa a ser definida como $\bar{P} = (1-\alpha)P + \alpha\frac{1}{n}U$ onde n é o número de vértices do grafo, α é a probabilidade de ocorrência dos saltos aleatórios e,

$$U = \begin{bmatrix} 1 & 1 & ... & 1 \\ 1 & 1 & ... & 1 \\ ... & ... & ... & ... \\ 1 & 1 & ... & 1 \end{bmatrix} \tag{2.5}$$

Desse modo, basta realizar o *Power Method* sobre a nova matriz e encontra-se a distribuição estacionária para a modelagem de um dígrafo.

2.1.4 Aplicação em Snakes and Ladders

Uma vez definido os conceitos de Cadeia de Markov e os métodos Power Method e Pagerank pra encontrar a distribuição estacionária após um tempo k, podemos responder perguntas como: Qual a chance do primeiro jogador vencer? Quantos movimentos são necessários para que algum jogador ganhe? Quais casas são mais prováveis de serem acessadas?

Para isso, é necessário modelar o tabuleiro de *Snake and Ladders* na forma de um grafo e extrair a matriz P. Portanto, tanto as abordagens utilizadas quanto os resultados e discussões são apresentadas na Seção 4.1

2.2 Problema 2: Árvore Geradora Mínima

Seja $G = (V, E)$ um grafo básico simples, dizemos que $T = (V', E')$ é uma Árvore Geradora se T é uma árvore que contém todos os vértices de G. Já uma Árvore Geradora Mínima é uma árvore árvore cuja soma do peso das arestas seja o menor possível. Portanto, o problema de extração de uma MST (Minimum Spanning Tree) refere-se ao problema onde, dado um grafo $G = (V, E, w)$ onde w refere-se ao peso das arestas, o objetivo é encontrar uma árvore $T = (V', E', w')$ de forma que a soma do peso das arestas seja o menor possível. Formalmente, $w(T) = \sum_{e \in T} w(e)$.

Existem diversos algoritmos capazes de realizar a extração de uma MST de modo eficiente, como, por exemplo, o algoritmo de Kruskal e o algoritmo de Prim que é discutido neste trabalho.

2.2.1 Algoritmo de Prim

O algoritmo de Prim, publicado por Robert C. Prim em 1957 (e por E. W. Dijkstra pouco depois) consiste em, dado um grafo $G = (V, E)$ com custo nas arestas e um vértice raiz r, produzir uma árvore geradora mínima cuja soma dos pesos das arestas é a menor possível.

O algoritmo de Prim faz crescer uma subárvore até que ela se torne geradora e usa como estrutura de dados auxiliar uma *Fila de Prioridades*. Para isso, o primeiro passo é atribuir para cada vértice $v \in V$ de G (com exceção da raiz r) o maior valor possível (infinito) como o menor custo de entrada e o vértice predecessor de v como *nulo*. Já a raiz, por ser a fonte de onde a árvore é gerada, é definida como custo de acesso 0 (sendo assim, o primeiro vértice a sair da fila de prioridades durante o laço principal). Por fim, adiciona-se na fila de prioridades Q todos os vértices do grafo G.

O crescimento da árvore ocorre durante o laço principal. Enquanto a fila de prioridades não estiver vazia, é extraido o vértice u com menor custo de acesso durante a iteração i e adicionado ao conjunto S de vértices finalizados. Em seguida, para cada vértice v vizinho de u que ainda está em Q busca-se o predecessor com menor custo de entrada que, ao ser encontrado, passa a ser predecessor de v que, por sua vez, tem seu custo de acesso atualizado.

O pseudocodigo para o algoritmo de Prim é apresentado abaixo.

Algorithm 1 Algoritmo de Prim para a geração de uma MST

```
function MST-Prim(G, w, s)
    for each v ∈ V[G] do
        v.chave = ∞
        v.π = NIL
    s.chave = 0
    Q = V[G]
    S = ∅
    while Q ≠ ∅ do
        u = ExtractMin(Q)
        S = S ∪ {u}
        for cada v ∈ N(u) do
            if v ∈ Q and v.chave > w(u, v) then
                v.chave = w(u, v)
                v.π = u
```

O Algoritmo de Prim pode ser aplicado em diversas situações práticas modeladas por grafos, como, por exemplo, extrair uma MST de um grafo cujos vértices representam cidades e as arestas ponderadas representam o custo/distância de uma cidade à outra.

Portanto, tanto a aplicação do algoritmo quanto os resultados e discussões feitas são apresentadas na Seção 4.2

2.3 Problema 3: Busca em Largura e Profundidade

O problema de busca em Teoria dos Grafos, refere-se a navegação em um grafo de determinística e sistemática alcançando todos os vértices. Consequentemente, buscar elementos em um grafo pode ser visto como um processo de extração de uma árvore $T = (V', E')$ em um grafo $G = (V, E)$.

Dois algoritmos principais são utilizados quando se trata de busca em grafos, são eles: Busca em Largura e Busca em Profundidade.

2.3.1 Busca em Largura

O algoritmo de Busca em Largura ou BFS (do inglês, Breadth-First Search) é um dos algoritmos mais básicos e intuitivos para se realizar busca em um grafo, seja ele direcionado ou não. Além disso, é o algoritmo modelo para diversos outros algoritmos importantes como, o Algoritmo de Prim e o Algoritmo de Dijkstra [CLS12].

A BFS possui uma ideia intuitiva bem simples. Dado um grafo $G = (V, E)$ e um vértice raiz r, a busca em largura explora todas as arestas de G calculando a menor distância (isto é, o menor número de arestas) entre r e todos os demais vértices presentes no grafo. O nomenclatura "em largura" se dá pelo fato de que se expande uniformemente entre os vértices descobertos e não descobertos ao longo do grafo. Isto é, o algoritmo descobre todos os vértices com distância k da raiz r antes de descobrir os vértices com distância $k + 1$.

Além disso, o algoritmo de busca em largura usa de uma Fila auxiliar e cores para representar os status dos vértices, sendo:

- WHITE: vértice não visitado, ou seja, vértice que ainda não entrou na fila Q
- GRAY: vértice já visitado, ou seja, está na fila Q
- BLACK: vértice finalizado, ou seja, já saiu da fila Q

Assim, o algoritmo inicializa todos os vértices (com exceção da raiz) com status WHITE, custo de acesso infinito (maior possível) e vértice predecessor nulo. Já o vértice raiz é inicializado com a status de descoberto (GRAY) custo 0, posto que, a raiz é o vértice de partida sendo assim, o primeiro a ser visitado e o primeiro a ser adicionado na fila Q.

Em seguida, inicia-se o laço principal onde é retirado da fila Q o vértice u (no caso da primeira iteração, u é a raiz) e busca-se todos os seus vizinhos. Para cada vizinho v de u, se v ainda não foi descoberto (WHITE), seu custo de acesso é atualizado onde, $v.d = u.d + 1$ (isto é, como v é descedente de u, então o custo de acesso a $v.d = u.d + 1$) e adiciona-se u como predecessor de v. Após a descoberta de todos os vizinhos de u, o mesmo é finalizado tendo seu status alterado para BLACK.

Ao final, o algoritmo terá produzido uma lista de predecessores, onde cada vértice $v \in V$ de G terá um predecessor correspodente. A árvore gerada pelo algoritmo da BFS é denominada "árvore

de busca em largura".

O pseudocodigo do método de busca em largura é apresentado abaixo. (Cormen, 2002).

Algorithm 2 Algoritmo de Busca em Largura

```
function BFS(G, s)
    for cada v ∈ V[G] − {s} do
        v.color = WHITE
        v.d = ∞
        v.π = NIL

    s.color = GRAY
    s.d = 0
    s.π = NIL
    Q = ∅
    PUSH(Q, s)

    while Q ≠ ∅ do
        u = POP(Q)
        for cada v ∈ N(u) do
            if v.color == WHITE then
                v.d = u.d + 1
                v.π = u
                PUSH(Q, v)
        u.color = BLACK
```

2.3.2 Busca em Profundidade

O Algoritmo de Busca em Profundidade, como o próprio nome diz, busca o mais "profundo" possível em um grafo. Em outras palavras, isso significa que o processo de descoberta de vértices ocorre explorando todas as arestas inexploradas do vértice v mais recentemente descoberto. Depois que todas as arestas que tem origem em v foram descobertas, a busca regressa pelo mesmo caminho e passa a explorar todas as arestas possíveis de serem exploradas que deu origem a v. Esse processo é feito continuamente até todos os vértices serem descobertos.

Além disso, em uma árvore de busca em profundidade (DFS Tree) diferente de uma árvore de busca em largura (BFS Tree) cujo o subgrafo de predecessores formam uma árvore, o subgrafo de predecessores de uma DFS pode ser composto por várias árvores pois a busca pode se repetir partindo de várias raízes. Desse modo, forma-se não apenas uma árvore, mas sim uma floresta de busca em profundidade que abrange várias árvores de busca em profundidade.

Outra propriedade importante da DFS é sua a caracteristica temporal de descoberta e

finalização de um vértice que nos permite ter noção tanto da profundidade dos vértices presentes no grafo.

Quanto ao algoritmo, a DFS possui um comportamento semelhante à BFS quando se diz respeito a estratégia utilizada para definir os status de um vértice. Isto é,

- WHITE: vértice não visitado, ou seja, vértice que ainda não entrou na fila Q
- GRAY: vértice já visitado, ou seja, está na fila Q
- BLACK: vértice finalizado, ou seja, já saiu da fila Q

Já para suprir a propriedade temporal, a DFS define dois atributos, um responsável por caracterizar tempo de descoberta do vértice v ($v.d$) e um que caracteriza o tempo de finalização ($v.f$) que é registrado quando a busca termina de analisar a lista de adjacencias do vértice v.

Logo, assim como ocorre com a BFS, o algoritmo inicializa todos os vértices com o status WHITE, custo de acesso infinito e vértice predecessor nulo.

Em seguida, é realizado o ajuste do contador de tempo e a chamada para método DFS-Visit(G, v) é feita. O vértice v enviado como parâmetro se torna a raiz de uma nova árvore na floresta de profundidade. Em cada chamada de DFS-Visit(G, u), o vértice u é inicialmente WHITE, portanto, atualiza-se o tempo contido na variavel global, o tempo de descoberta de u para o tempo atual da variavel global e a cor do vértice agora que o mesmo foi descoberto.

Adiante, para cada vértive v vizinho do vértice u em questão, caso v não tenha sido descoberto, v tem seu predecessor atualizado e é feita a chamada para método DFS-Visit(G, u) onde v é passado como parâmetro. Com isso, inicia-se um processo recursivo vértice a vértice. Ao final, quando todos os vértices possíveis forem descobertos a partir de u, seu status é atualizado para BLACK (finalizado) assim como, o tempo da variavel global e o tempo de finalização. Ao fim do processo, o resultado é uma floresta de busca em profundidade a partir do mapa de predecessores.

O pseudocodigo do método de busca em profundidade é apresentado abaixo. (Cormen, 2002).

Algorithm 3 Algoritmo de Busca em Profundidade

```
function DFS(G, s)
    for cada v ∈ V[G] do
        v.color = WHITE
        v.π = NIL

    time = 0
    DFS-VISIT(G, u)
```

Os algoritmos de busca em grafos podem ser aplicados em diversos casos do mundo real. Por exemplo, dado os grafos *Karete Club* e *Dolphins Network* extrair tanto uma árvore de busca em largura quanto uma árvore de busca em profundidade de ambos. Desse modo, tanto a aplicação dos algoritmos quanto os resultados e discussões feitas são apresentadas na Seção 4.3

```
function DFS-VISIT(G, u)
    time = TIME + 1
    u.d = TIME
    u.color = GRAY

    for cada v ∈ N(u) do
        if v.color == WHITE then
            v.π = u
            DFS-VISIT(G, v)

    u.color = BLACK
    time = TIME + 1
    u.f = TIME
```

2.4 Problema 4: Árvores de Caminhos Mínimos e Agrupamento de Dados

A extração de árvores de caminhos mínimos em grafos ponderados é um problema extremamente relevante em Teoria dos Grafos e imensamente útil para diversas aplicações práticas. Dado um grafo ponderado $G = (V, E, w)$ onde w representa o peso das arestas, extrair uma árvore de caminhos mínimos $T = (V', E', w')$ de G significa encontrar uma árvore cujo peso do caminho entre dois vértices (u, v) seja o menor possível.

Apesar de parecer um problema simples e semalhante aos problemas de busca e extração de árvores geradoras mínimas, o problema de extração de caminhos mínimos em grafos ponderados possui diversas variantes:

- **Problema de caminhos mínimos de fonte única**: Encontrar um caminho mínimo de um determinado vértice de origem r a todo vértice v pertencente ao grafo.
- **Problema de caminhos mínimos com um só destino**: Encontrar um caminho mínimo até um determinado vértice de destino t a partir de cada vértice v.
- **Problema do caminhos mínimos para um par**: Encontrar um caminho mínimo entre dois vértices u e v.
- **Problema de caminhos mínimos para todos os pares**: Encontrar um caminho mínimo entre u e v para todo par (u, v) do grafo.

Contudo, ao solucionar o problema de caminhos mínimos de fonte única podemos solucionar analogamente todas as outras variantes. Devido a isso, existem diversos algoritmos capazes de extrair uma árvore de caminhos mínimos de modo eficiente. O algoritmo apresentado e discutido neste trabalho é o Algoritmo de Dijsktra.

2.4.1 O Algoritmo de Dijkstra

O algoritmo de Dijkstra é o mais comum, utilizado e eficiente para os problemas de caminhos mínimos. Sua função é, dado um grafo $G = (V, E, w)$ ponderado (w) e uma raiz r (ou mais raizes), retornar uma árvore de caminhos mínimos (ou uma floresta de caminhos mínimos para o caso de mais de uma raíz), a menor distância do vértice raiz r a todos os demais vértices v do grafo.

O Algoritmo de Dijkstra é muito parecido com Algoritmo de Prim e de Busca em Largura. A estrutura de dados auxiliar utilizada é uma fila de prioridades (como Prim utiliza) para organizar os vértices de forma que, quanto menor o custo de acesso a um vértice v, maior a prioridade do vértice. Já na execução do laço principal, a execução do Algoritmo de Dijkstra é semelhante a Busca em Largura, isto é, a árvore se expande gradativamente pelos menores ramos (ramos com menor custo de acesso) na expectativa de que os caminhos mínimos mais longos usarão como base os subcaminhos obtidos anteriormente.

O algoritmo de Dijkstra mantém um conjunto S de vértices cujos pesos finais de caminhos mínimos que partem da raiz r já foram determinados. O algoritmo seleciona repetidamente o vértice $v \in V - S$ que possui o menor custo de acesso, adiciona u a S e relaxa todas as arestas que saem de u. O processo de relaxamento de uma aresta (u, v) consiste em testar se é possível melhorar o caminho mínimo até v encontrado até agora através de u. Caso sim, então tanto o custo de acesso a v $(v.d)$ e seu predecessor $(v.\pi)$ são atualizados.

O pseudocodigo do algoritmo de Dijkstra é apresentado abaixo. (Cormen, 2002).

Algorithm 4 Algoritmo para geração de uma árvore de caminhos mínimos

```
function DIJKSTRA(G, w, s)
    for cada v ∈ V[G] do
        v.d = ∞
        v.π = NIL

    s.d = 0
    Q = V[G]
    S = ∅

    while Q ≠ ∅ do
        u = EXTRACTMIN(Q)
        S = S ∪ {u}
        for cada v ∈ N(u) do
            if v.d > u.d + w(u, v) then
                v.d = u.d + w(u, v)
                v.π = u
```

O Algoritmo de Dijkstra pode ser utilizado em diversas situações no mundo real relacionado

a distâncias, por exemplo. Imagine que você vá viajar e deseja encontrar a rota mais curta entre a cidade de origem até a cidade de destino, onde no caminho há diversas outras cidades pelas quais você deve passar. Tal problema, pode ser interpretado como um problema de caminhos mínimos, onde cada cidade é um vértice e peso das arestas entre os vértices é a distância entre as cidades.

A fim de analisar a capacidade e funcionamento do Algoritmo de Dijkstra, o mesmo foi aplicado sobre um grafo cujos vértices representam 50 cidades da Alemanha Ocidental e as arestas, o custo de acesso entre elas. Os resultados e discussões feitas são apresentadas na Seção 4.4

2.5 Problema 5: Sudoku Problem Solver Com Coloração De Vértices

Embora o *Sudoku* seja comumente associado à cultura japonesa e oriental, ele foi originalmente desenvolvido pelo americano Howard Garns com o nome de *Number Place* e publicado pela primeira vez no final da década de 70 em revistas especializadas em desafios e quebra-cabeças.

O jogo consiste em um tabuleiro 9x9 subdividido em nove boxes 3x3. O tabuleiro vem inicialmente preenchido com alguns números que variam de 1 a 9. O objetivo do jogador é, portanto, completar o restante das células. No entanto não podem haver números repetidos na mesma linha, coluna ou box. O nível de dificuldade do jogo varia conforme a quantidade de números previamente preenchidos.

Atualmente existem diversas variações do jogo, como por exemplo, com tabuleiros 4x4, 16x16, ou até mesmo 9x9 em que não pode haver repetições dos números nas diagonais principais. Mesmo com todas essas as variações, ainda é possível resolver o *Sudoku* usando coloração de vértices, onde cria-se um grafo de incompatibilidade, em que células que estão na mesma linha, coluna ou box serão vértices que estarão conectados através de uma aresta. Após a criação do grafo, o problema é facilmente resolvido através do algoritmo *Welsh & Powell*. Resolver o *Sudoku* pela força bruta seria computacionalmente menos eficiente e demorado, já que existem 6.670.903.752.021.072.936.960 (6.7 mil bilhões) maneiras de se preencher o tabuleiro inicial 9x9.

2.5.1 Coloração de vértices

O principal objetivo da Coloração de Vértices é rotular vértices com "cores", de tal de forma que não haja dois vértices adjacentes que compartilhem da mesma cor, sendo assim, eles formem conjuntos independentes.

O algoritmo mais utilizado para resolver o problema da Coloração de vértices é o algoritmo de Welsh & Powell, que tem como propósito sempre minimizar o número de cores necessárias para colorir o grafo.

Ele consiste em, após criar o grafo de incompatibilidade, ordenar os vértices em ordem decrescente de grau. Após, atribuir uma lista de cores a cada vértice de forma que quanto maior o grau do vértice, menor a lista, sendo que o número máximo de elementos da lista de cores não pode ultrapassar o valor do vértice de grau máximo.

Depois disso feito, o algoritmo entra em um *loop* onde é atribuída ao vértice a cor da primeira cor da lista de cores. Em seguida, é retirado da lista de cores a cor de todos os vértices que são adjacentes ao vértice cuja cor esta sendo atribuída.

O pseudocódigo do algoritmo de Welsh & Powell é apresentado abaixo.

Algorithm 5 Algoritmo para coloração de vértices

```
function WELSH & POWELL(G)
    Ordenar os vértices {x_1, x2, ..., x_n} | Grau(x_i) ≥ Grau(x_i + 1)
    for i = 2 to N do
        C_i = {}
    C_1 = {x_1}
    for i = 2 to N do
        k = MIN {j | Adj(x_i) Inter Cj = {}, j = 0, 1, ..., n}
        C_k = C_k ∪ {x_j}
    return MAX { j | C_j <> {}, j = 1, 2, ..., n}
```

3. Materiais e Métodos

3.1 Bibliotecas e Pacotes

Todos os experimentos e soluções foram implementadas através da linguagem *Python (v 3.6.6)*. As bibliotecas utilizadas em conjunto da linguagem foram:

- **numpy (v 1.15)**: Pacote fundamental para computação científica em Python que permite trabalhar e manipular *arrays* N-dimensionais com extrema facilidade e eficiência. O pacote *numpy* contém também uma quantidade relevante de funções sofisticadas e otimizadas para cálculos de matrizes, transformadas de Fourier, etc.
- **networkx (v 2.2)**: Como descrito na Seção 1.6, a biblioteca *networkx* é um pacote para a criação, manipulação e estudo de grafos e redes complexas, sejam essas estruturas estáticas ou dinâmicas. A biblioteca fornece uma grande quantidade de algoritmos para grafos já implementados de forma eficiente, além de, fornecer diversas formas de criação de grafos, dígrafos e qualquer tipo de estrutura complexa necessária.
- **matplotlib (v. 3.0)**: A biblioteca *matplotilib* é um pacote Python para *plotagem* de figuras em diversos variedades, formatos e ambientes cuja qualidade seja o suficiente para publicações. A partir do pacote é possível gerar gráficos de dispersão, histogramas, espectros de potência, gráficos de barras, gráficos de erros e muitos outros.

3.2 Ambiente de Desenvolvimento

Os códigos foram implementados através do editor de texto extensível (e IDE) *Visual Studio Code* (também denominado, *vscode*. Em sua versão *default* o editor *vscode* possui suporte para depuração para linguagens como Python, C, C++, Java e muitas outras. Além disso, conta também com controle de versões Git incorporado, *syntax highlighting* (realce de sintaxe), complementação

inteligente de código, *snippets* e refatoração de código. Contudo, através de extensões e configurações de usuário é possível complementar diversas das funcionalidades e transforma-lo em um ambiente de desenvolvimento completo para diversas linguagens de programação ou marcação diferentes.

Além do editor de texto, a plataforma de controle de versão *GitHub* também foi utilizada para armazenamento, controle de versão e atualização dos códigos conforme estes foram sendo desenvolvidos. A plataforma é muito utilizada para divulgação de projetos, trabalhos e contribuições da comunidade através de ferramentas de comunicação como *issues* e *pull requests*.

Por fim, o sistema operacional (SO) utilizado tanto para o desenvolvimento quanto execução dos experimentos foi a distribuição Linux Ubuntu 18.04 LTS. O Ubuntu é um sistema operacional baseado em Linux produzido pela empresa sul-africana Canonical. O SO é totalmente gratuito e se baseia na filosofia Ubuntu, ou seja, o software deve ser disponibilizado gratuitamente, as ferramentas de software devem ser utilizáveis por qualquer pessoa (deficiente ou não) em suas línguas locais, além de ser permitido ao usuário personalizar e alterar o software de qualquer maneira que o atenda.

3.3 Conjuntos de Dados

Os conjuntos de dados utilizados para os experimentos foram 4 no total.

- **30 Cidades Pelo Mundo (HA30.ZIP) e Árvores Geradoras Mínimas**: Conjunto de dados que contém 30 cidades ao redor do mundo. As informações são divididas em dois arquivos principais. O arquivo HA30_NAME.ZIP contém os nomes das cidades pertencentes aos dados e o arquivo HA30_DIST.TXT contém uma matriz de adjacências representam a distância de cada uma das cidades para todas as outras. Esses dados foram utilizados para o problema de Árvores Geradoras Mínimas;
- **Zachary's Karate Club (KARATE.PAJ) e Busca em Largura e Profundidade**: A *Zachary's Karate Club* é a rede social de um clube universitário de karatê descrita no artigo *An Information Flow Model for Conflict and Fission in Small Groups* de Wayne W. Zachary. A rede, que pode ser modelada com um grafo, contém 34 vértices que representam cada um dos membros do clube e as arestas que representam as relações fora do clube entre os membros. Tal conjunto de dados foi utilizado para aplicação dos algoritmos de Busca em Largura e Busca em Profundidade.
- **Dolphins Network (DOLPHINS.PAJ) e Busca em Largura e Profundidade**: A *Dolphins Online Social Network* representa uma rede social de golfinhos. Cada vértice do grafo refere-se a um membro da comunidade de golfinhos que vivem em Doubtful, Nova Zelândia. As arestas representam relações frequentes entre um par de golfinhos. Os dados referem-se a observações realizadas a partir o ano de 1994 até 2001.
- **59 Cidades Alemãs (WG59.ZIP) e o Algoritmo de Dijkstra**: O arquivo WG59.ZIP representa um conjunto de dados contendo 59 cidades da Alemanha ocidental. Assim, cada vértice do grafo representa uma cidade e as arestas o respectivo custo de acesso de uma cidade à outra.

4. Resultados e Discussões

4.1 Solução 1: Snakes and Ladders

Como discutido na Seção 2.1, a aplicação dos conceitos de Cadeia de Markov e *Power Method* requer que o tabuleiro seja modelado como um grafo. Portanto, a variação de *Snakes and Ladders* escolhida foi a versão com 36 casas (grade 6x6) (como mostra a Figura 4.1) cujas regras de navegação são:

- Em cada rodada um jogador joga uma moeda não viciada e avança 1 casa se obtiver cara ou avança 2 casas se obtiver coroa.
- Se o jogador para no pé da escada, então ele imediatamente sobe para o topo da escada.
- Se o jogador cai na boca de um cobra então ele imediatamente escorrega para o rabo.
- O jogador sempre inicia no quadrado de número 1.
- O jogo termina quando ele atinge o quadrado de número 36.

Assim, dada as regras do jogo, podemos modelar o tabuleiro como um dígrafo onde cada vértice i será ligado aos vértices $i+1$ e $i+2$. Os vértices que representam casas do tabuleiro com uma cabeça de cobra são ligados ao vértice que representa o rabo daquela cobra. O mesmo é feito para as escadas.

Computacionalmente, isso pode ser feito como apresenta o código.

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

def setGraph():
    G = nx.DiGraph()
    G.add_edges_from([(x, y) for x in range(1,36) for y in range(x+1,x+3) if y!= 37])

    G.add_edge(5, 7)
```

```
    G.add_edge(2, 15)
    G.add_edge(9, 27)
    G.add_edge(18, 29)
    G.add_edge(25, 35)

    G.add_edge(34, 12)
    G.add_edge(32, 30)
    G.add_edge(24, 16)
    G.add_edge(26, 6)
    G.add_edge(17, 4)

    return G
```

Listing 4.1: Modelando Dígrafo

Figure 4.1: Tabuleiro de *Snake and Ladders* com 36 casas

O que nos resulta no dígrafo apresentado pela Figura 4.2

Na definição padrão da Cadeia de Markov, a matriz de probabilidade de transição de estados P é defina como $P = \Delta^{-1}A$. Portanto, em posse do dígrafo, basta extrairmos a matriz de adjacências A, definirmos a matriz Δ^{-1} e então, computamos P.

```
def main():
    G = setGraph()
    A = nx.to_numpy_matrix(G)
    delta = np.zeros((36, 36))
    np.fill_diagonal(delta,[1/float(G.degree(i)) for i in range(1, 37) \
```

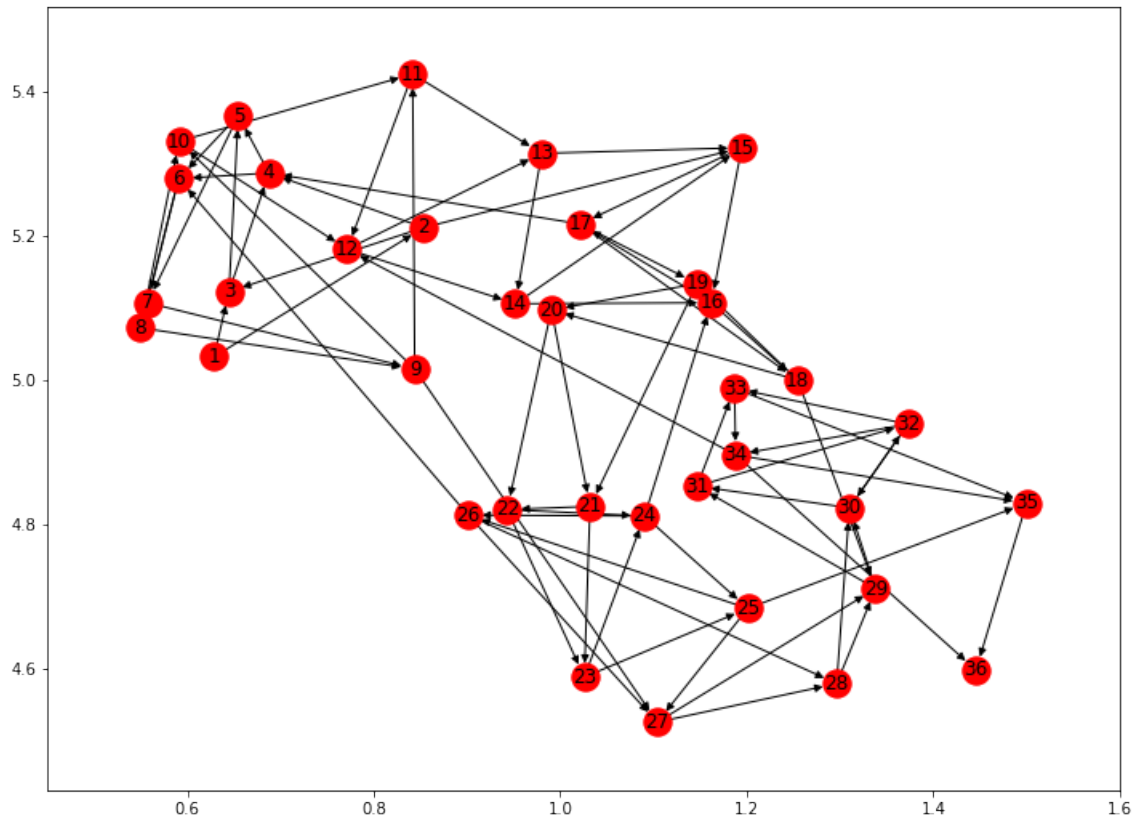


Figure 4.2: Dígrafo referente ao tabuleiro

```
    if G.degree(i)!=0])
    P = np.dot(delta, A)
```

Listing 4.2: Calculo da matriz P

Enfim, basta calcularmos a distribuição estacionária $w^{(k)}$ através do *Power Method*. É importante observar, no entanto, que o vetor w para $k = 0$ deve ter seu primeiro índice inicializado com 1, pois é onde o jogo começa (casa 1).

```
def powerMethod(P):
    w =  np.zeros(36)
    w[0] = 1

    for i in range(100):
        w = np.dot(w, P)

    probs = {}
    w = w.tolist()
    w = w[0]
    for i in range(0, len(w)):
        probs[i+1] = w[i]
```

```
    return probs
```

Listing 4.3: Calculo da matriz P

Adotando $k = 100$ como número de iterações para o *Power Method* encontramos uma distribuição estacionária da forma $w^{(100)} =$ [0.00E+00, 0.00E+00, 0.00E+00, 1.63E-35, 7.06E-36, 2.67E-35, 1.54E-35, 1.99E-35, 1.91E-35, 1.91E-35, 1.86E-35, 4.35E-35, 2.89E-35, 3.45E-35, 3.44E-35, 5.24E-35, 3.76E-35, 3.90E-35, 3.32E-35, 3.49E-35, 3.69E-35, 3.89E-35, 4.10E-35, 4.33E-35, 4.10E-35, 3.65E-35, 4.19E-35, 3.40E-35, 5.35E-35, 6.62E-35, 5.19E-35, 5.68E-35, 5.27E-35, 5.32E-35, 6.94E-35, 6.06E-35]

Note que nesse caso, a probabilidade de se atingir o estado 36 é de 6.06e-35. Porém, apesar de ser um número bem pequeno, o estado 36 ainda continua sendo um dos cinco estados com maior probabilidade do jogador estar após 100 iterações, sendo o cinco estados apresentados pela Tabela 4.1

Estado	Probabilidade
35	6.94E-35
30	6.62E-35
36	6.06E-35
32	5.68E-35
29	5.35E-35

Table 4.1: Cinco maiores probabilidades para $k = 100$

Contudo, é importante ressaltar que como discutido na Seção 2.1, a Cadeia de Markov padrão para Dígrafos tende a não expressar os resultados da melhor forma possível. Por isso, a matriz P deve ser definida da forma $\bar{P} = (1-\alpha)P + \alpha\frac{1}{n}U$ onde n é o número de vértices do grafo, α é a probabilidade de ocorrência dos saltos aleatórios, permitindo assim uma melhor convergência por parte do modelo.

Portanto, adotando $\alpha = 0.1$, calculamos a matriz $\bar{P}$

```
def main()
    #[...]
    U = np.ones((36,36))
    P_barra = np.dot((1-0.1), P)+ np.dot((0.1*(1/36)), U)
```

Listing 4.4: Calculo da matriz $\bar{P}$

Em seguida, utilizamos do *Power Method* para encontrar a distribuição estacionária, agora para a matriz $\bar{P}$.

```
def powerMethod(P_barra):
    w =  np.zeros(36)
    w[0] = 1
```

```

    for i in range(100):
        w = np.dot(w, P_barra)

    probs = {}
    w = w.tolist()
    w = w[0]
    for i in range(0, len(w)):
        probs[i+1] = w[i]

    return probs
```

Listing 4.5: Calculo da matriz P

Com isso, encontramos uma distribuição estacionária bem diferente da anterior. Neste caso, a probabilidade do jogador estar após 100 iterações na casa 36 é de 1.70E-29. Os cinco estados com maiores probabilidades são exibidos pela Tabela 4.2

Estado	Probabilidade
16	1.83E-29
30	1.81E-29
35	1.70E-29
29	1.69E-29
12	1.69E-29

Table 4.2: Cinco maiores probabilidades para $k = 100$

Um resultado muito importante que deve ser ressaltado é fato de que, apesar das distribuições estacionárias para P e $\bar{P}$ serem completamente diferentes, os estados (casas) 29, 30 e 35 estão entre os cinco estados com maior probabilidade em ambos os casos.

4.1.1 Repositórios e Códigos-fonte

O repositório com o código-fonte completo pode ser encontrado em: `https://github.com/a-hayasi/graphtheory2018`

4.2 Solução 2: Árvore Geradora Mínima

Como discutido na Seção 2.2, o problema da extração de uma árvore geradora mínima pode ser representado em diversos casos no mundo real. Por exemplo, dado um grafo cujos vértices representam cidades e as arestas o custo para ir de uma cidade à outra, qual árvore geradora mínima encontrariamos ? Para responder essa pergunta, basta aplicarmos o Algoritmo de Prim no grafo e encontrarmos a MST.

O conjunto de dados `ha30.zip` contém o nome de 30 cidades em todo mundo, assim como, o custo de acesso de cada uma. Podemos então, importar a matriz que define o custo para ir de uma cidade à outra como a matriz de adjacências do grafo que representará as cidades e suas respectivas conexões e, a partir disso, montar o grafo e chamar o Algoritmo de Prim para o grafo com raíz no vértice 0

```
def main():
    A = np.loadtxt('ha30_dist.txt')
    G = nx.from_numpy_matrix(A)
    mst = prim(G.copy(), 0)
```

Listing 4.6: Montando o grafo

A implementação do Algoritmo de Prim na linguagem Python (v. 3.6.6) pode ser feita da seguinte maneira.

```
1  def prim(G, r):
2      Q = {}
3      for v in G:
4          attrs = {v: {'key': np.inf, 'p': None}}
5          nx.set_node_attributes(G, attrs)
6          Q[v] = G.nodes[v]
7
8      G.nodes[r]['key'] = 0
9      S = []
10
11     while Q:
12         min_weight = min([Q[i]['key'] for i in Q])
13         u = [i for i in Q if Q[i]['key'] == min_weight][0]
14         S.append(u)
15         del Q[u]
16         for v in G.neighbors(u):
17             if v in Q and G.nodes[v]['key'] > G.get_edge_data(u, v)['weight']:
18                 G.nodes[v]['key'] = G.get_edge_data(u, v)['weight']
19                 G.nodes[v]['p'] = u
```

Listing 4.7: Algoritmo de Prim

Assim como ocorre com o pseudocódigo, todos os vértices de G são inicializados com o maior custo de acesso possível e predecessor nulo (linhas 3, 4 e 5). A fim de facilitar a implementação ao mesmo tempo em que o código fosse o mais didático possível, a fila de prioridades Q é definida como um dicionário onde, todos os vértices pertencentes ao grafo são alocados (linha 6). Em seguida, a raiz é inicializada com custo igual a 0 (afinal, deve ser o primeiro vértice a sair da fila de prioridades), o conjunto de vértices finalizados S é instaciado e inicia-se o laço principal.

As linhas 12, 13 e 15 do Listing 4.7 representam uma implementação simplificada do método EXTRACTMIN(Q) presente no pseudocódigo enquanto que a linha 14 é responsável por adicionar o vértice u removido de Q ao conjunto de vértices finalizados S.

A partir da linha 16 em diante, ocorre o processo de busca das arestas com menor custo. Isto é, para cada vértice $v \in Q$ vizinho de u, se o custo de acesso a v atual é maior do que o custo para acessar v através de u, então tanto o custo de acesso a v quanto seu predecessor são atualizados, do mesmo modo que é feito no pseudocódigo.

Por fim, como o resultado do Algoritmo de Prim é um mapa de predecessores, montamos o grafo a partir do conjunto de vértices finalizados S e seus predecessores e, retornamos a árvore geradora mínima encontrada.

```
def prim(G, r):
    #[...]
    mst = nx.Graph()
    for v in S:
        if G.nodes[v]['p'] != None:
            mst.add_edge(v, G.nodes[v]['p'], \
            weight=G.get_edge_data(v, G.nodes[v]['p'])['weight'])

    return mst
```

Listing 4.8: Retorno da MST gerada

O código abaixo serve para verificar o número de vértices, arestas e o peso total de G e a MST gerada pelo Algoritmo de Prim.

```
def main():
    #[...]
    print('numero de vertices de G: {:d}\n'
          'numero de arestas  de G: {:d}\n'
          'peso de G:  {:.1f}\n'.format(G.number_of_nodes(),
                                        G.number_of_edges(),
                                        G.size(weight='weight')))

    print('numero de vertices da MST: {:d}\n'
          'numero de arestas  da MST: {:d}\n'
          'peso da MST gerada: {:.1f}\n'.format(mst.number_of_nodes(),
                                                mst.number_of_edges(),
                                                mst.size(weight='weight')))
```

Listing 4.9: Retorno da MST gerada

A saída do código é apresentado na Tabela 4.3

Grafo	Num. vértices	Num. arestas	Peso
G	30	435	24710
MST	30	29	403

Table 4.3: Resultado do Algoritmo de Prim para G

Note que, o número de arestas do grafo é como esperado pela teoria. Isto é, $|V| - 1$. Além disso, o peso total da árvore geradora mímina 61 vezes menor que o grafo original. A MST gerada pode ser visualizada na Figura 4.3

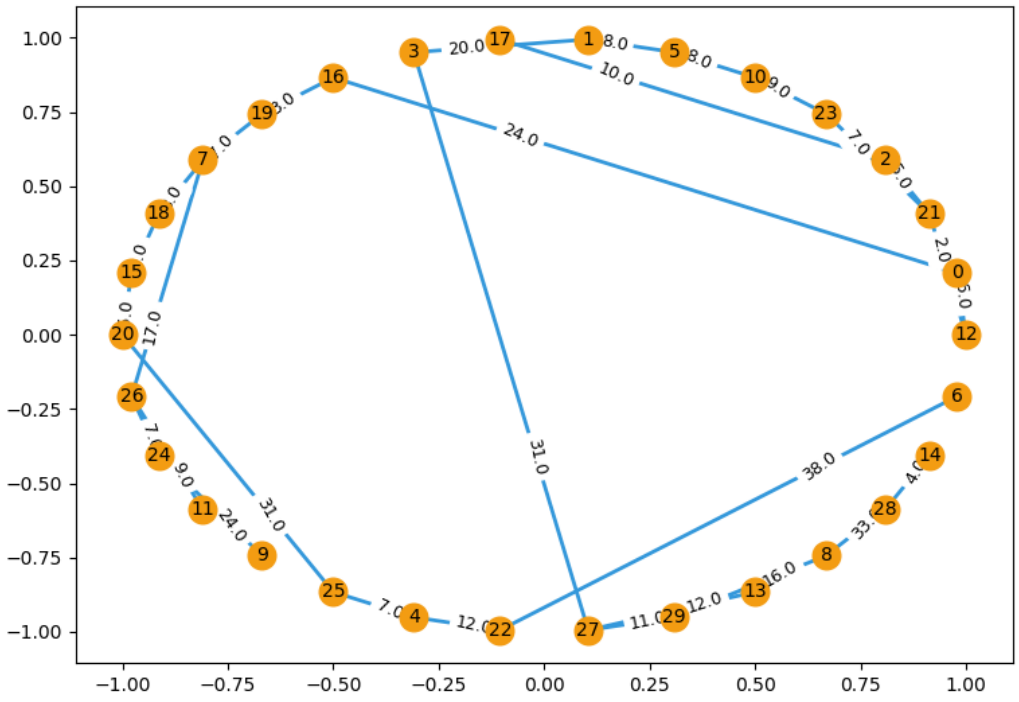


Figure 4.3: MST gerada através do Algoritmo de Prim

4.2.1 Repositórios e Códigos-fonte

O repositório com o código-fonte completo pode ser encontrado em: `https://github.com/a-hayasi/graphtheory2018`

4.3 Solução 3: Busca em Largura e Profundidade

4.3.1 Busca em Largura

Busca em Largura e Busca em Profundidade são dois algoritmos amplamente aplicados em diversas situações e problemas relacionados a busca de elementos em grafos. Por exemplo, dado os grafos *Zachary's karate club* e *Dolphins*, podemos extrair uma árvore de busca em largura e uma árvore de busca em profundidade sobre seus vértices? Para responder essa pergunta, devemos aplicar ambos algoritmos de busca em ambos os grafos.

No entanto, o primeiro passo é importar para o programa os grafos em questão. Através da linguagem Python, esse processo pode ser feito de forma bem simples.

```
def main():
    G1 = nx.read_pajek('karate.paj')
    G2 = nx.read_pajek('dolphins.paj')
```

Listing 4.10: Retorno da MST gerada

O algoritmo de Busca em Largura implementado na linguagem Python é de fácil reconhecimento (assim como o Algoritmo de Prim). Como apresentado no Listing 4.11, a implementação é idêntica ao pseudocódigo. Isto é, primeiro são inicializado os vértices com status (color), custo de acesso (d) e predecessor (p), enquanto que o vértice raiz é inicializado com a cor *gray* (visitado) e custo de acesso 0.

Em seguida, a raíz é adicionada à fila Q e inicia-se o laço principal cujo funcionamento é o mesmo descrito no pseudocódigo.

```
def bfs(G, r):
    for v in G:
        attrs = {v: {'color': 'white', 'd': np.inf, 'p': None}}
        nx.set_node_attributes(G, attrs)

    G.nodes[r]['color'] = 'gray'
    G.nodes[r]['d'] = 0
    Q = []
    Q.append(r)

    while Q:
        u = Q.pop(0)
        for v in G.neighbors(u):
            if G.nodes[v]['color'] == 'white':
                G.nodes[v]['d'] = G.nodes[u]['d'] + 1
                G.nodes[v]['p'] = u
                G.nodes[v]['color'] = 'gray'
                Q.append(v)
        G.nodes[u]['color'] = 'black'
```

Listing 4.11: Algoritmo de Busca em Largura

Enfim, assim como ocorre com o Algoritmo de Prim, a árvore de Busca em Largura é feita a partir dos vértices e seus respectivos predecessores encontrados durante o algoritmo.

```
    bfs_tree = nx.Graph()
    for v in G:
        if G.nodes[v]['p'] != None:
            bfs_tree.add_edge(G.nodes[v]['p'], v)

```

```
    return bfs_tree
```

Listing 4.12: Montagem da árvore de Busca em Largura

As Figuras 4.4 e 4.5 apresentam as árvores de Busca em Largura encontradas após a aplicação do algoritmo sobre os grafos *Zachary's karate club* e *Dolphins*, respectivamente.

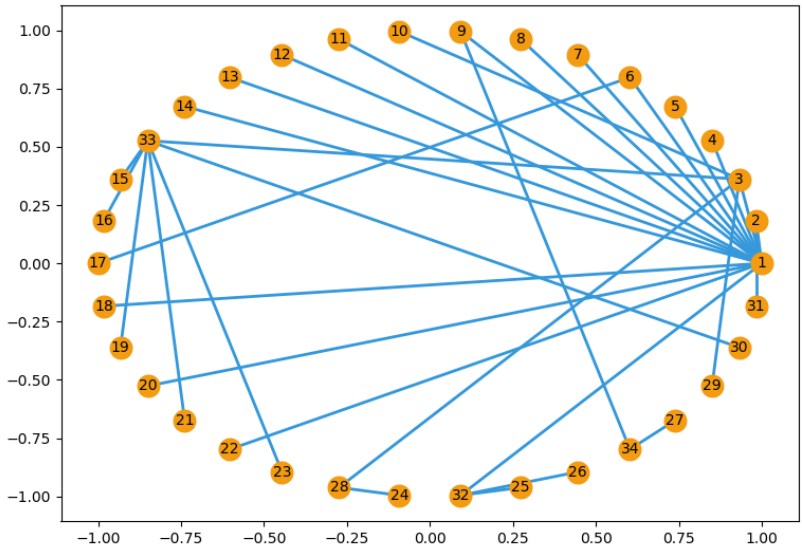


Figure 4.4: BFS Tree *Zachary's karate club*

4.3.2 Repositórios e Códigos-fonte

O repositório com o código-fonte completo pode ser encontrado em: `https://github.com/a-hayasi/graphtheory2018`

4.3.3 Busca em Profundidade

Para a aplicação da Busca em Profundidade, o mesmo procedimento utilizado para a Busca em Largura foi adotado, cuja única diferença está, realmente, nos algoritmos.

Contudo, do mesmo modo que foi com os outros algoritmos, a Busca em Profundidade implementada na linguagem Python é idêntica ao pseudocódigo apresentado na Seção 2

```
def dfs(G, r):
    for v in G:
        attrs = {v: {'color': 'white', 'p': None, 'd': None, 'f':None}}
        nx.set_node_attributes(G, attrs)

    global time
```

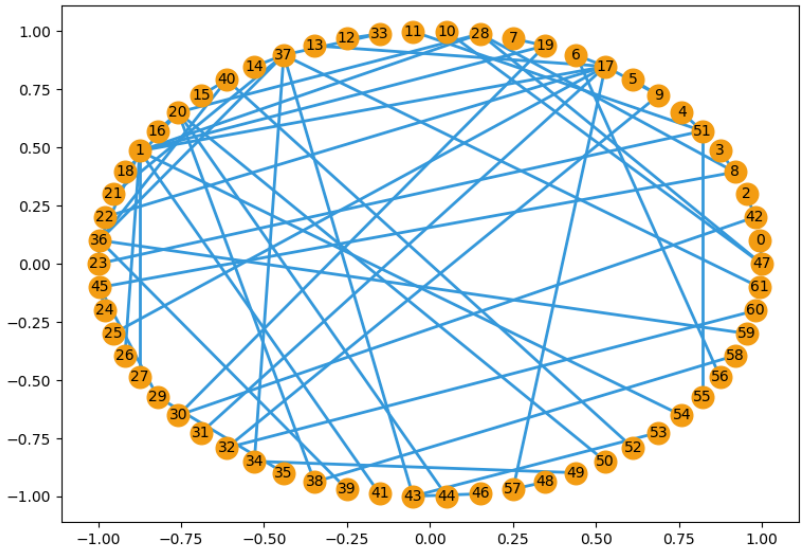


Figure 4.5: BFS Tree *Dolphins*

```
    time = 0
    dfs_visit(G, r)


def dfs_visit(G, u):
    global time
    time += 1
    G.nodes[u]['d'] = time
    G.nodes[u]['color'] = 'gray'

    for v in G.neighbors(u):
        if G.nodes[v]['color'] == 'white':
            G.nodes[v]['p'] = u
            dfs_visit(G, v)

    G.nodes[u]['color'] = 'black'
    time += 1
    G.nodes[u]['f'] = time
```

Listing 4.13: Algoritmo de Busca em Profundidade

Como a floresta de busca em profundidade pode ser construída através do mapa de predecessores definido, então o mesmo procedimento adotado para os demais algoritmos também é aplicado para a DFS.

```
def dfs(G, r):
    #[...]
    dfs_tree = nx.Graph()
    dfs_tree.add_node(r)

    for v in G:
        if G.nodes[v]['p'] != None:
            dfs_tree.add_edge(G.nodes[v]['p'], v)

    return dfs_tree
```

Listing 4.14: Algoritmo de Busca em Profundidade

As Figuras 4.6 e 4.7 apresentam as florestas de busca em profundidade encontradas após a aplicação do algoritmo sobre os grafos *Zachary's karate club* e *Dolphins*, respectivamente.

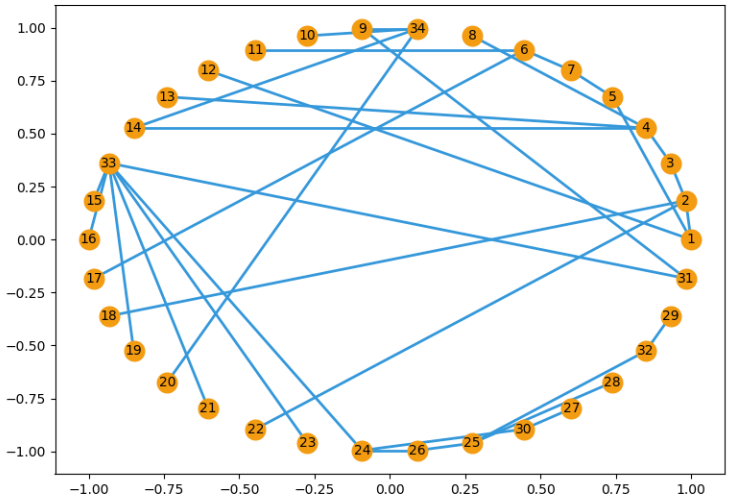


Figure 4.6: DFS Tree *Zachary's karate club*

4.3.4 Repositórios e Códigos-fonte

O repositório com o código-fonte completo pode ser encontrado em: `https://github.com/a-hayasi/graphtheory2018`

4.4 Solução 4: Árvores de Caminhos Mínimos e Agrupamento de Dados

Suponha que você deseja encontrar a rota mais curta possível entre Berlim e Bonn (Alemanha). Dado um mapa rodoviário da Alemanha no qual a distância entre cada par de interseções adjacentes esteja marcado, como determinar a rota mais curta?

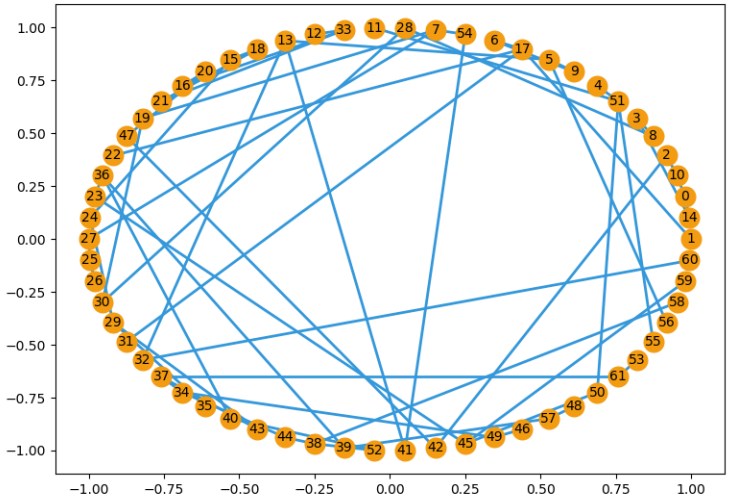


Figure 4.7: DFS Tree *Dolphins*

Existem diversas distâncias, rotas e ciclos possíveis o que torna o trabalho exaustivo cujo resultado pode não ser o melhor possível. Por isso, podemos modelar tal problema como um problema de extração de caminhos mínimos e aplicar o algoritmo de Dijkstra.

Assim, dado um grafo cujos vértices representam 59 cidades da Alemanha ocidental e as arestas o respectivo custo de acesso de uma cidade à outra, qual o caminho mínimo entre uma cidade de origem (vértice raiz) até todas as demais cidades? Basta aplicarmos Dijkstra!

Computacionalmente, o primeiro passo é importar a matriz de adjacências que expressa o grafo, transformá-lo em um grafo e enfim, aplicar o algoritmo de Dijkstra.

Duas aplicações iniciais foram feitas, primeiro com duas raízes (0, 28) e três raízes (0, 28 e 58).

```
def main():
    A = np.loadtxt('wg59_dist.txt')
    G = nx.from_numpy_matrix(A)
    graph1 = dijkstra(G, [0, 28])
    graph2 = dijkstra(G, [0, 28, 58])
```

Listing 4.15: Algoritmo de Busca em Profundidade

O algoritmo de Dijkstra implementado na linguagem Python é, assim como os outros algoritmos apresentados até agora, idêntico ao pseudocódigo. A principal diferença está no fato de que em Dijkstra, mais de uma raiz é passada, portanto, mais de um vértice é inicializado com custo de acesso igual a 0.

```
def dijkstra(G, raizes):
```

```
    Q = {}
    for v in G:
        attrs = {v : {'d': np.inf, 'p': None}}
        nx.set_node_attributes(G, attrs)
        Q[v] = G.nodes[v]

    for s in raizes:
        G.nodes[s]['d'] = 0

    S = []
    while Q:
        min_weight = min([Q[i]['d'] for i in Q])
        u = [i for i in Q if Q[i]['d'] == min_weight][0]
        S.append(u)
        del Q[u]

        for v in G.neighbors(u):
            aux = G.nodes[u]['d'] + G.get_edge_data(u, v)['weight']
            if  G.nodes[v]['d'] > aux:
                G.nodes[v]['d'] = aux
                G.nodes[v]['p'] = u
```

Listing 4.16: Algoritmo de Busca em Profundidade

Da mesma forma, podemos construir a floresta de caminhos mínimos através do mapa de predecessores definido pelo algoritmo.

```
def dijkstra(G, raizes):
    #[...]
    dijkstra_tree = nx.Graph()
    for v in S:
        if G.nodes[v]['p'] != None:
            dijkstra_tree.add_edge(G.nodes[v]['p'], v, \
            weight = G.get_edge_data(G.nodes[v]['p'], v)['weight'])

    return dijkstra_tree
```

Listing 4.17: Algoritmo de Busca em Profundidade

As Figuras 4.8 e 4.9 apresentam as florestas de caminhos mínimos encontradas após a aplicação do algoritmo de Dijkstra sobre o grafo das cidades alemãs.

4.4.1 Repositórios e Códigos-fonte

O repositório com o código-fonte completo pode ser encontrado em: `https://github.com/a-hayasi/graphtheory2018`

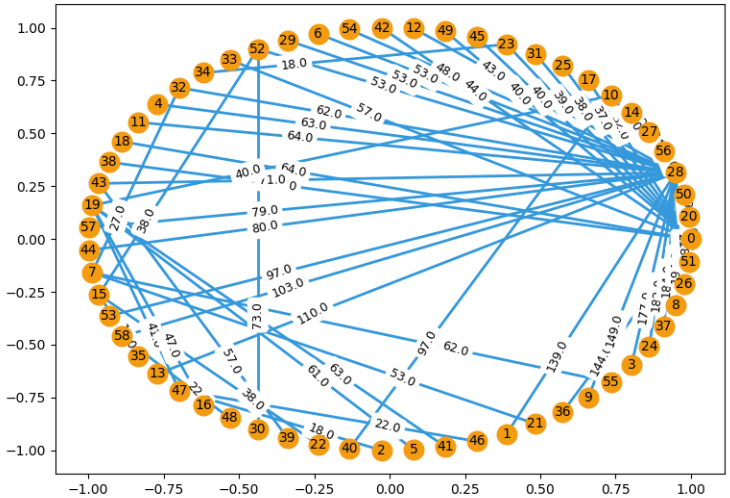


Figure 4.8: Floresta de Caminhos Mínimos com raiz no vértice 0 e 28

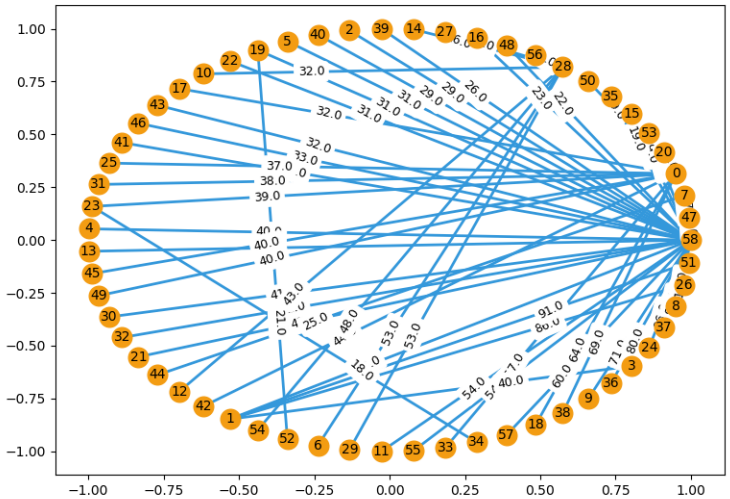


Figure 4.9: Floresta de Caminhos Mínimos com raiz no vértice 0, 28 e 58

4.5 Solução 5: Sudoku Problem Solver Com Coloração De Vértices

Como discutido na Seção 2.5, o primeiro passo para que se possa resolver o *Sudoku* usando coloração de vértices é criar um grafo de incompatibilidade. Neste caso, após a criação de um grafo com 81 vértices correspondendo a cada célula do tabuleiro, uma aresta é adicionada ao grafo se:

- Se um vértice vi pertence a mesma linha que um vj
- Se um vértice vi pertence a mesma coluna que um vértice vj
- Se um vértice vi pertence ao mesmo box que um vértice vj

Em seguida é mostrada a função que adiciona arestas ao grafo:

```
def add_aresta(tabuleiro, grafo):
    for i in range (0,9):
        for j in range (0,9):
            if tabuleiro[i][j] == 0:
                for k in range(0,9):
                    if tabuleiro[i][k] != 0:
                        grafo.add_edge(i+9*j, i+9*k)
                    if tabuleiro[k][j] != 0:
                        grafo.add_edge(i+9*j, k+9*j)

                for k in range(0,3):
                    for l in range(0,3):
                        auxi = i//3
                        auxj = j//3
                        if tabuleiro[auxi+k][auxj+l] != 0:
                            grafo.add_edge(i+9*j,i+k+9*(j+l))
```

Listing 4.18: Função que adiciona arestas ao grafo

Após, na função que implementa o algoritmo de Welsh & Powell, o vértice de maior grau do tabuleiro e que possui rótulo 0 é encontrado. Desta forma, somente vértices que ainda não possuem coloração serão selecionados.

Posteriormente, este vértice uma "cor", no caso um número de 1 a 9. A função retorna -1 -1 caso não encontre vértice que possua rotulação 0, pois isto indica que não há mais células a serem coloridas e o algoritmo pode ser encerrado.

```
def welshPowell(tabuleiro, grafo):
    maior_grau = 0
    auxi = -1
    auxj = -1
    for i in range(0,9):
        for j in range(0,9):
            if len(list(grafo.neighbors(i+9*j))) > maior_grau and tabuleiro[i][j] ==
    0:
                maior_grau = len(list(grafo.neighbors(i+9*j)))
                auxi = i
                auxj = j
    return auxi, auxj
    return -1, -1
```

Listing 4.19: Implementação Welsh & Powell

Por fim, a função recursiva resolveSudoku chama a função que cria o grafo de incompatibilidade para que este seja atualizado a cada passo da recursão. Ela inicializa com i e j = 0, e assim chama a função Welsh & Powell atribuindo seus valores ao i e j.

Caso o valor do i seje igual a -1, ela então sabe que o tabuleiro está completamente preenchido e retorna *True*. Caso ainda falte número para preencher, a função faz um *loop* de 1 a 10 e chama uma função que verifica se o valor é válido e atribui a ele o valor de i em caso positivo.

Em seguida, para que todos os valores do tabuleiro sejam preenchidos, a função se chama recursivamente e só para quando retornar *True*. Se isto não ocorrer significa que não há mais possibilidades de números a serem preenchidos e a função retorna *False*

```
def resolveSudoku(tabuleiro, grafo, i=0, j=0):
    add_aresta(tabuleiro, grafo)
    i, j = welshPowell(tabuleiro, grafo)
    if i == -1:
        return True
    for e in range(1, 10):
        if verificaSeValorEValido(tabuleiro, i, j, e):
            tabuleiro[i][j] = e
            if resolveSudoku(tabuleiro, grafo, i, j):
                return True
    return False
```

Listing 4.20: Função que resolve o Sudoku

4.5.1 Questionamentos

Após a implementação do algoritmo, é possível responder aos questionamentos propostos:

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

Caso o algoritmo ache uma solução válida os graus de todos os vértices serão iguais a 20 (8 da linha, 8 da coluna e 4 restante do box). Se o algoritmo não encontrar uma solução os graus do vértice pode variar de acordo com o tabuleiro.

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

A solução encontrada foi:

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

As soluções são iguais, pois para um mesmo tabuleiro, o grafo de incompatibilidade criado será sempre o mesmo, assim como sua lista de graus.Desta forma, como o algoritmo sempre seleciona a primeira cor da lista de graus e atribui ao vértice, não importa quantas vezes o algoritmo for rodado, a solução será sempre a mesma.

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.

```
Problema Resolvido:

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
-----------------------
```

Figure 4.10: Solução do Tabuleiro apresentado

e) O que acontece se a pré-coloração inicial tiver apenas 9 valores (1 por linha)?

Para ambas as questões, não foi possível encontrar uma solução válida, pois para que um *Sudoku* possua uma única solução válida, ele deverá ter no mínimo 28 células preenchidas inicialmente.

4.5.2 Repositórios e Códigos-fonte

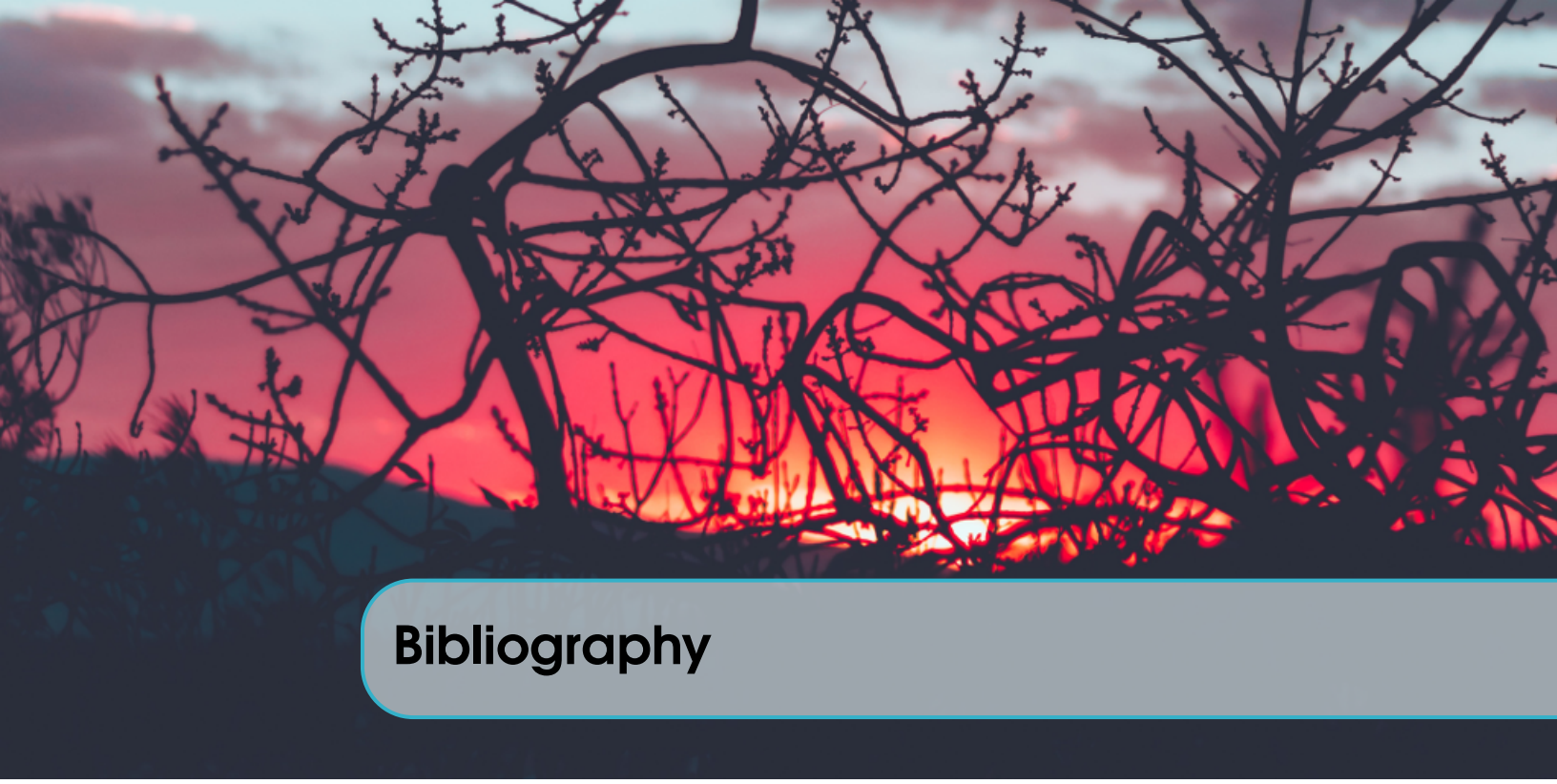
O repositório com o código-fonte completo pode ser encontrado em: `https://github.com/a-hayasi/graphtheory2018`

5. Considerações Finais

A aplicação de Teoria dos Grafos para a resolução de problemas práticos é extremamente eficiente, eficaz e satisfatória. Apesar da denominação "teoria", a Teoria dos Grafos é essencial para diversos aspectos práticos do mundo real e pode ser aplicada na resolução de variados tipos de problemas. Desde questões como movimentação, locomoção e trajetos até estabelecimento de relações, alocação de recursos, tarefas e projeto de sistemas, a Teoria dos Grafos fornece uma gama de conhecimento gigantesco para sempre encontrar a melhor solução possível.

Além disso, a linguagem de programação *Python* possui um enorme poder resolução de tarefas complexas através da implementação dos algoritmos de grafos em nível computacional pois, tanto os diversos pacotes e bibliotecas existentes, como *networkx* e *numpy* quanto a facilidade de implementação de certos códigos, tornam tanto a experiência quanto eficiência na solução de problemas extremamente satisfatória.

Portanto, a linguagem de programação *Python* é imprescindível para a implementação e execução prática tanto dos problemas quanto dos algoritmos da Teoria dos Grafos, seja para motivos didáticos ou de aplicação real.



Bibliography

[1] Algoritmos: teoria e prática, Cormen, T.H. and Leiserson, C.E. and Stein, R.L.R.E.C. 9788535236996,2012,ELSEVIER EDITORA

[2] A First Look at Graph Theory. A Clark, J. A Holton, D.A. 9789810204907. 1991. I World Scientific

[3] Graph Theory and Complex Networks: An Introduction. van Steen, M. 9789081540612. 2010. Maarten van Steen

[4] A.A. Markov. "Extension of the limit theorems of probability theory to a sum of variables connected in a chain". reimpresso no Apêndice B de: R. Howard. Dynamic Probabilistic Systems, volume 1: Markov Chains. John Wiley and Sons, 1971.

[5] Networkx Developers. Disponível em https://networkx.github.io/. Acesso em Dezembro de 2018.

[6] Numpy Developers. Disponível em http://www.numpy.org/. Acesso em Dezembro de 2018.

[7] Matplotlib Developers. Disponível em https://matplotlib.org. Acesso em Dezembro de 2018.

[8] Python Developers. Disponível em https://www.python.org/. Acesso em Dezembro de 2018.

[9] Wikipedia. Disponível em https://en.wikipedia.org/wiki/Snakes_and_Ladders. Acesso em Dezembro de 2018.

[10] Wikipedia. Disponível em https://en.wikipedia.org/wiki/Zachary%27s_karate_club. Acesso em Dezembro de 2018.

[11] Network Repository. Disponível em http://networkrepository.com/soc-dolphins.php. Acesso em Dezembro de 2018.