UNIVERSIDADE FEDERAL FLUMINENSE

ELIAQUIM DOS SANTOS MAURICIO

AYLTON VIEIRA DA SILVA NAZÁRIO

ALGORITMOS EM GRAFOS E SISTEMAS INTELIGENTES

Niterói

2016

ELIAQUIM DOS SANTOS MAURICIO
AYLTON VIEIRA DA SILVA NAZÁRIO

ALGORITMOS EM GRAFOS E SISTEMAS INTELIGENTES

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Niterói, 05 de Janeiro de 2017.

Banca Examinadora:

_____

Prof. Luis Antonio Brasil Kowada, D. Sc. – Orientador

UFF – Universidade Federal Fluminense

_____

Prof. Luís Felipe Ignácio Cunha, M. Sc. – Orientador

UFRJ – Universidade Federal do Rio de Janeiro

_____

Prof. Fernanda Vieira Dias Couto, D. Sc. – Avaliador

UFRRJ – Universidade Federal Rural do Rio de Janeiro

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

M455 Mauricio, Eliaquim dos Santos
Algoritmos em grafos e sistemas inteligentes / Eliaquim dos Santos Mauricio, Aylton Vieira da Silva Nazário. – Niterói, RJ : [s.n.], 2016.
57 f.

Projeto Final (Tecnólogo em Sistemas de Computação) – Universidade Federal Fluminense, 2016.
Orientador: Luis Antonio Brasil Kowada.

1.Sistemas inteligentes (Ciência da computação). 2. Algoritmo em grafo. 3. Drone. I. Nazário, Aylton Vieira da Silva. II. Título.

CDD 006.33

AGRADECIMENTOS

A Deus, que sempre iluminou a nossa caminhada.

A nossa família e aos nossos orientadores em especial ao Luís Felipe Ignácio Cunha pela paciência, estímulo e atenção que nos concedeu durante todo o curso.

Aos colegas de curso pelo incentivo e troca de experiências.

“O coração do sábio
se inclina para o bem,
mas o coração do tolo, para o mal.”
Eclesiastes 10:2

RESUMO

Este trabalho disserta sobre sistemas de Drones, complexidade de algoritmos, algoritmos em grafos e uma aplicação de grafos ao controle de Drones.

Existe uma necessidade muito grande de softwares para Drones, devido suas diversas aplicações, mesmo a curto prazo, como por exemplo entrega de produtos ou em eventos de grande multidão onde o Drone é usado para filmagem. O Google e a Amazon têm tocado projetos bem-sucedidos com este tipo de aeronaves.

Apesar dos testes estarem indo bem ainda existem problemas a serem resolvidos, por exemplo, a organização do espaço aéreo, a duração de baterias, o alcance do controle e a incapacidade de detectar objetos inesperados. Sendo assim, este último problema será estudado, propomos uma estratégia para desvio de obstáculos, com o objetivo de possivelmente contribuir para uma futura solução do mesmo.

Palavras-chaves: Algoritmos em grafos, Desvio de obstáculos, Drone.

ABSTRACT

This document discusses Drones systems, complexity of algorithms, algorithms in graphs and an application of graphs on controlling Drones.

There is a very large usage of Drones for many applications, even in a short time of existence, such as product delivery or large crowd events where Drones are used for filming. Google and an Amazon have touched on successful projects with this type of aircraft.

Although the tests are going well, there are still problems to be solved, for example, the organization of airspace, the duration of batteries, the reach of control and the inability to detect unexpected objects. Therefore, this last problem will be studied, we propose a strategy for the diversion of obstacles, with the aim of contributing for a future solution of this problem.

Key words: Algorithms in graphs, Detour of obstacles, Drone.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

VANT - Veículo aéreo não tripulado

API - Application Programming Interface

MAVLink - Micro Air Vehicle Link

MST - Minimal Spanning Tree

Graphviz - Graph Visualization Software

DFS - Depth-first search

BFS - Breadth-first search

SUMÁRIO

1. INTRODUÇÃO

1.1 DRONES

O termo Drone é usado para referenciar qualquer veículo voador que não precisa de piloto embarcado, são veículos aéreos não tripulados (VANTs). Eles são controlados a distância através de eletrônicos e computadores tendo ou não a supervisão de humanos.

Figura 1 - Exemplo de Drone.

Atualmente, em vários países (principalmente nos EUA), há um desenvolvimento intenso de pesquisas e fabricação de Drones. Inclusive, há vários anos, vem sendo um dos principais instrumentos da estratégia militar.

No Brasil existem projetos criados para desenvolver esses veículos com diversas finalidades, por exemplo, serem usados para o auxílio da segurança pública, monitoramento ambiental e de trânsito, telecomunicações, entre outros. Já as forças armadas brasileiras pretendem utilizá-los na vigilância das fronteiras e do mar territorial (Contribuidores, Veículo aéreo não tripulado, 2016).

O controle de navegação de um Drone normalmente é feito por sensores como uma bússola magnética e um GPS. Para desvio de obstáculos algum tipo de sonar é usado. Alguns Drones utilizam também outros mecanismos como por exemplo, uma ou mais câmeras para fazerem o mapeamento da área e desviar de potenciais obstáculos.

1.1.1 A ‘API’ DRONEKIT

O Dronekit (Contributors, 3D Robotics, 2015) é uma API em linguagem Python, ou seja, uma biblioteca com diversos artefatos de software utilizados no domínio de aplicação de Drones. Esta API permite o acesso ao sistema do Drone com suas interfaces bem definidas, permitindo o envio e recebimento de informações através de comandos de cada componente da API utilizando seu protocolo de comunicação MAVLink. Tudo isso auxilia na criação de aplicativos para Drones, além disso, é um projeto código aberto e totalmente dirigido pela comunidade. Os aplicativos criados com essa API rodam em computadores que fazem a comunicação com a placa controladora de voo usando o protocolo de comunicação MAVLink.

Atualmente, há uma rica documentação na internet, que provê material guia e demonstrações, incluindo referências. Algumas características desta API podem ser vistas abaixo:

- Conexão com um ou múltiplos veículos a partir de um script.
- Obter e atribuir estado/telemetria e informações de parâmetro do veículo.
- Receber notificações assíncronas de modificação de estado.
- Orientar o VANT a uma posição especificada (Modo Guiado).
- Enviar mensagens customizadas de forma arbitrária para controlar o movimento e outros hardwares do VANT
- Criar e gerir missões com rotas geradas por pontos definidos no mapa. (Modo automático)

1.2 A LINGUAGEM PYTHON

A maioria dos Drones do mercado trabalha com a linguagem Python ou C++, alguns utilizam a extensão Dronekit (vista anteriormente). O Python é uma linguagem de programação de alto nível, interpretada, de script, imperativa, orientada

a objetos e de tipagem dinâmica e forte. Um conteúdo mais completo pode ser visto no livro (Tanner & Brueck, 2001) e (Python Software Foundation, 2016).

Estas características são descritas abaixo:

- É uma linguagem de *alto nível*, ou seja, o programador se abstrai de características de componentes do computador, como por exemplo, instruções e registradores usados pelo processador, ou então o uso de memória.
  (Wikipedia, High-level programming language, 2016)
- Seu modo de execução é *interpretado*, sendo assim, não é necessário o estágio de compilação para a execução do código, a sintaxe do código é lida e executada diretamente. Basicamente, o programa faz uma chamada ao interpretador que lê o bloco de código do programa seguindo seu fluxo, e então decide o que fazer e o faz. Caso haja algum erro no código ele apenas será identificado em tempo de execução diferentemente de linguagens compiladas onde erros são identificados durante o estágio de compilação.
  (Wikipedia, List of programming languages by type, 2016)
- É uma *linguagem de script* que em uma descrição informal é um roteiro de código utilizado para automatizar alguma tarefa.
  (Wikipedia, List of programming languages by type, 2016)
- É uma *linguagem imperativa*, onde instruções são escritas para realizarem alguma ação de forma que mudem o estado do programa. Pode ser feita uma analogia com o modo imperativo em linguagens naturais onde ele é usado para expressar comandos. Em um programa imperativo, os comandos são enviados para o computador realizá-los.
  (Wikipedia, Imperative programming, 2016)
- Possui um paradigma *orientado a objetos*, onde os objetos são criados de acordo com sua classe que define suas características e comportamento, chamados de atributos e métodos respectivamente. Os dados e os métodos para manipulá-los são mantidos na unidade chamada de objeto, como boa prática de programação, outros objetos só devem acessar seus dados através dos métodos do objeto.
  (Wikipedia, Programming paradigm, 2016)

- É uma linguagem de programação imperativa que suporta uma linguagem *funcional*, esta linguagem funcional avalia funções matemáticas e evita mudanças de estado dos dados e mudanças nos objetos após a sua criação. (Wikipedia, Functional programming, 2016)
- Possui *tipagem dinâmica*, onde são executados muitos comportamentos comuns de linguagens de programação em tempo de execução. Exemplos desses comportamentos são: Adicionar novo código, estender objetos, mudar tipos do sistema como, por exemplo, tipos de dados. Em linguagens de tipagem estática esses comportamentos são realizados durante a compilação. (Wikipedia, Dynamic programming language, 2016)
- É uma linguagem dita de *tipagem forte*, ou seja, os tipos de dados (caracter, numérico, booleano, etc.), devem ser respeitados em parâmetros de funções e atribuição de dados a variáveis, pois se forem passados valores de tipos diferentes possivelmente será gerado um ou vários erros. (Wikipedia, Strong and weak typing, 2016)

Exemplo 1: Um pequeno trecho de código em Python que imprime a sequência de Fibonacci pode ser visto abaixo:

```
def fibonacci(n):    # Escreve a sequência de Fibonacci no intervalo de 0 á n
    num1, num2 = 0, 1
    while num1 < n:
        print(num1, end=' ')
        auxnum = num1
        num1 = num2
        num 2 = auxnum + num2
    print()

 # Uma chamada a função com n=500
fibonacci(500);
# resultado: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

1.3 BIBLIOTECAS PYTHON PARA GRAFOS

1.3.1 GRAPH-TOOL

Graph-tool é uma ferramenta gratuita em Python para manipulação e análise estatística em grafos (assunto que será visto com mais detalhes na Seção 1.3), sua estrutura de dados e algoritmos são implementados em C/C++ e fazem grande uso do modelo de metaprogramação (Wikipedia, Metaprogramming, 2016) e da Biblioteca Boost Graph (The Boost Graph Library, 2003). Muitos dos algoritmos foram implementados utilizando a API OpenMP (Wikipedia, OpenMP, 2016), que permite o processamento paralelo, ou seja, mais de uma thread em execução, isso se torna muito útil e agiliza o processamento de algoritmos que podem ser “quebrados” em partes e distribuídos em múltiplos núcleos para que sejam processados em paralelo e assim o tempo de execução seja então otimizado. Possui diversas características como algoritmos topológicos (MST, componentes conexos, etc.), geração de grafos aleatórios, estatísticas de grafos, entre outras. É bem documentado, incluindo vários exemplos. Possui também uma boa usabilidade com seus algoritmos de layout, e pode também trabalhar de forma integrada com o graphviz, um software que permite a visualização de Grafos utilizando diagramas como forma de ilustração das informações estruturais do grafo que são passadas como parâmetro. Para maiores detalhes sobre o Graph-tool, consultar (What is graph-tool?).

1.3.2 IGRAPH

Igraph é uma coleção de ferramentas de análise sobre grafos, é de código aberto e gratuito, pode ser programado nas linguagens R, Python, C/C++. A biblioteca é relativamente simples de utilizar, há repositórios no GitHub com suas bibliotecas e pacotes incluindo documentações. Maiores informações em (The igraph core team, 2015).

1.3.3 NETWORKX

NetworkX é um pacote de ferramentas para criação, manipulação e estudo de grafos. Promete ser uma ferramenta de alta produtividade para redes complexas, na qual alguns objetivos estão descritos abaixo:

O estudo e análise estatística das características das redes, como a disposição de graus entre vértices ou o tamanho do principal componente conexo do grafo. (Dahis, 2009)

Formalizar os modelos de redes para melhor entendimento do processo de formação e organização de tais redes estudadas e analisadas. (Dahis, 2009)

Criação de modelos que sejam capazes de prever o comportamento desses sistemas, baseados nas propriedades de cada vértice ou de toda a rede. (Dahis, 2009)

Compreender através do estudo e análise estatística, como esta conectividade de vértices (objetos) através de arestas (relacionamento), atua sobre a funcionalidade e os processos relacionados a estes vértices. (Figueiredo,, 2011)

A *NetworkX* é rica em documentação, exemplos e referências. Possui diversos colaboradores desenvolvedores com o repositório no GitHub. Sua implementação foi feita puramente em Python.

Algumas características do NetworkX são:

- Estruturas de dados de grafos, dígrafos e multigrafos na linguagem Python.
- Várias unidades de teste.
- Código aberto.
- Muitos algoritmos padrões de grafos.
- Apoio à análise com medidas.

Mais detalhes em (NetworkX developer team, 2016).

1.3.4 COMPARAÇÃO DE PERFORMANCE DAS BIBLIOTECAS

Uma comparação de performance é apresentada no site do graph-tool (Graph-tool performance comparison, 2015). Nesta comparação foram utilizadas as três ferramentas com suas respectivas versões, Graph-tool 2.11, Igraph 0.7.1 e NetworkX 1.10. Foram utilizados alguns algoritmos e testados sobre o mesmo grafo $G = (V, E), onde\ V = 39{,}796\ e\ E = 301{,}498$ (Esse grafo é direcionado e fortemente conexo). Os algoritmos foram executados diversas vezes utilizando um processador Intel (R) Core (TM) i7-5500U CPU @ 2.40GHz com 4 núcleos, a Figura 2 mostra os algoritmos com suas médias de execução em cada uma das ferramentas. Para o graph-tool cada um dos algoritmos foi executado em um momento utilizando os 4 núcleos e em outro momento utilizando apenas um núcleo (multi-threads e single-thread).

Algorithm	graph-tool (4 cores)	graph-tool (1 core)	igraph	NetworkX
Single-source shortest path	0.004 s	0.004 s	0.012 s	0.152 s
PageRank	0.029 s	0.045 s	0.093 s	3.949 s
K-core	0.014 s	0.014 s	0.022 s	0.714 s
Minimum spanning tree	0.040 s	0.031 s	0.044 s	2.045 s
Betweenness	244.3 s (~4.1 mins)	601.2 s (~10 mins)	946.8 s (edge) + 353.9 s (vertex) (~ 21.6 mins)	32676.4 s (edge) 22650.4 s (vertex) (~15.4 hours)

Figura 2 - Comparação de três bibliotecas Python.

É notável a diferença de tempo para algoritmos que são executados mais rapidamente utilizando paralelismo, como exemplo é possível observar o algoritmo *PageRank* que executado com 1 núcleo teve um tempo de 0,045s, quando executado com 4 núcleos atingiu o tempo de 0,029s. Já o algoritmo *Minimum spanning tree* obteve um tempo a mais de 0,09s sendo executado com 4 núcleos comparado a execução com 1 núcleo apenas.

Além da comparação de performance neste cenário, existem outros fatores a serem avaliados antes da escolha de uma das bibliotecas, isto depende muito de cada caso de uso. A ferramenta graph-tools pode não ser considerada vantajosa para determinada situação pelo grande tempo de compilação e utilização de memória, ou

então o problema de o sistema operacional não ter um pré-compilador de binário disponível. A ferramenta Igraph possui um tempo de compilação menor que a Graph-tools, já a ferramenta NetworkX não requer compilação (pelo fato de ser puramente implementada em Python) e pode ser iniciada rapidamente, para grafos não muito grandes ela pode ser vantajosa. Cada ferramenta possui suas próprias características, sua própria API, conjunto de algoritmos implementados, além de suas vantagens e desvantagens que devem ser consideradas de acordo com vários fatores como os expressados aqui, além de muitos outros. O conteúdo deste parágrafo, do anterior e a Figura 2 foram adaptados de (Graph-tool performance comparison, 2015).

1.4 GRAFOS

A *teoria dos grafos* é um ramo da matemática que estuda as relações entre objetos de um determinado conjunto, para tal finalidade são utilizadas estruturas chamadas de grafos.

Grafos são usados para resolver problemas em diversos campos, tais como na representação de qualquer rede de rotas de transporte (um mapa de estradas, por exemplo), rede de comunicação (como em uma rede de computadores), ou rotas de distribuição de produtos ou serviços, como dutos de gás ou água, etc. A estrutura química de uma molécula também pode ser representada por um grafo. Algumas referências de estudo da teoria dos grafos são: (Wikipedia, Graph theory, 2016), (Jon & Éva, 2005), (Cormen, Charles, Ronald, & Clifford, 2002) e (Szwarcfiter & Markenzon, 2010).

1.4.1 CONTEXTO HISTÓRICO

O artigo de Leonhard Euler, publicado em 1736, sobre o problema das sete pontes de Königsberg, é considerado o primeiro resultado da teoria dos grafos.

O problema das sete pontes é baseado na cidade de Königsberg que é cortada pelo Rio Prególia, onde há duas grandes ilhas que, juntas, formam um complexo que na época continha sete pontes. Discutia-se nas ruas da cidade a possibilidade de atravessar todas as pontes sem repetir nenhuma. Havia-se tornado uma lenda popular a possibilidade da façanha quando Euler, em 1736, provou que não existia caminho que possibilitasse tais restrições.

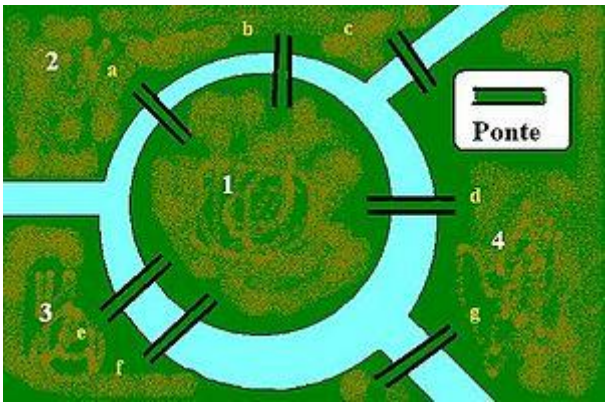

Figura 3 - Esquema das pontes em Königsberg.

Euler usou um esquema bastante simples, desenhou os caminhos em linhas (arestas do grafo) e suas interseções em pontos (vértices do grafo). Então percebeu que só seria possível atravessar o caminho inteiro passando uma única vez em cada ponte se houvesse exatamente zero ou dois pontos de onde saísse um número ímpar de caminhos.

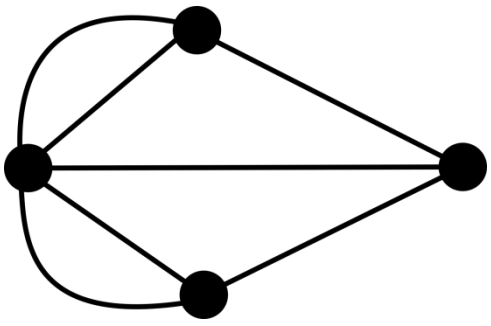

Figura 4 - Grafo estilizado das pontes.

1.4.2 DEFINIÇÕES

Formalmente, um *Grafo G* é um conjunto de três elementos {V(G), E(G),ψ_G}, onde V(G) é um conjunto não vazio e finito de *vértices* e $|V(G)| = n\ e\ |E(G)| = m$, sendo E(G) um conjunto disjunto de V(G) que representam as arestas e ψ_G sendo uma função incidente que associa com cada aresta de G um par não ordenado de vértices de G que não precisam ser distintos. Um exemplo pode ser visto abaixo (Ver Figura 5):

$V(G) = \{v1, v2, v3, v4, v5, v6\}$
V(E) = {e1, e2, e3, e4, e5, e6}
$\psi_G(e1) = v1\ v2, \quad \psi_G(e2) = v1\ v3, \quad \psi_G(e3) = v3\ v2, \ \psi_G(e4) = v4\ v2, \quad \psi_G(e5) = v4\ v5, \psi_G(e6) = v5\ v6,$

Os grafos podem ser ou não ser direcionados. Em grafos direcionados, as arestas possuem uma direção que é representada por uma seta indicando o vértice em que a aresta incide. Em grafos não direcionados as arestas não possuem direção e indicam um relacionamento mútuo. A Figura 5 mostra um exemplo de um grafo não direcionado e a Figura 6 mostra um exemplo de outro grafo, porém direcionado. Um estudo mais detalhado pode ser visto em J.A. Bondy and U.S.R. Murty, 1976.

Normalmente, utiliza-se uma representação gráfica (geométrica) de um grafo, como pode ser visto na Figura 5.

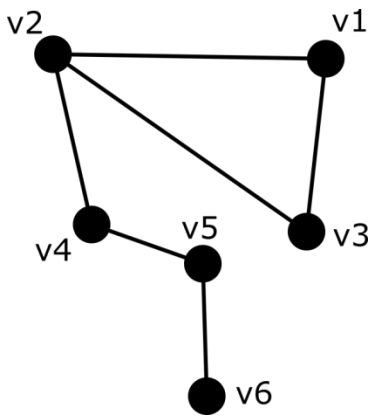

Figura 5 - Exemplo de um Grafo Não Direcionado com seis vértices e seis arestas.

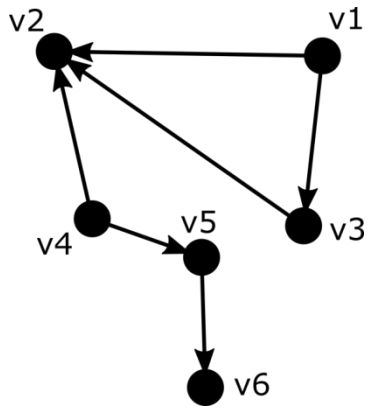

Figura 6 - Exemplo de um Grafo Direcionado com seis vértices e seis arestas.

Os grafos possuem alguns conceitos associados, como por exemplo:

Um *caminho* em um grafo G é dado por uma sequência não nula e finita $W = v_0 e_1 v_1 e_2 v_2 e_3 v_3 \dots e_k v_k$, onde v_k representa vértices e e_k arestas, para $1 \leq i \leq k$ e onde $v_0, v_1, v_2, v_3 \dots v_k$ são distintos. É comum se representar tal sequência utilizando apenas os vértices e representando as arestas pelas adjacências dos vértices, dessa forma W pode ser apresentado como $W = v_0 v_1 v_2 v_3 \dots v_k$, onde implicitamente existe uma aresta e_k entre $v_k e\, v_{k-1}, \forall\, i \mid 1 \leq i \leq k$. Caso $e_1, e_2, e_3 \dots e_k$ sejam distintos, então W é chamado de *trilha*. Se $v_0, v_1, v_2, v_3 \dots v_k$ não forem distintos entre si e

$e_1, e_2, e_3 \ldots e_k$ também não forem distintos entre si então W será chamado de *percurso*.

Um caminho é euleriano se o caminho passa por todas as arestas, sendo que utiliza cada aresta apenas uma vez, ao passo que um caminho é hamiltoniano se o caminho passa por todos os vértices e utiliza cada vértice apenas uma vez. Na Figura 7 vemos um exemplo de um caminho entre v_1 *e* v_6 no grafo G(V,E) | V=$\{v_1, v_2, v_3, v_4, v_5, v_6\}$ e E=$\{v_1v_2, v_1v_3, v_3v_2, v_2v_4, v_4v_5, v_5v_6\}$.

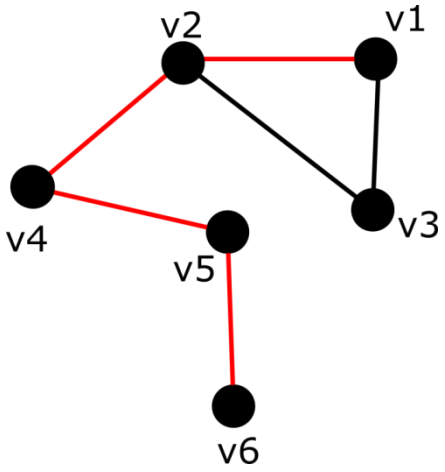

Figura 7 - Caminho v1 v2,v2 v4,v4 v5,v5 v6 em um Grafo G.

Ciclo é um um caminho que começa e termina no mesmo vértice, mas sem repetir vértices intermediários. Na Figura 5 existe o seguinte ciclo: v_1v_2, v_2v_3, v_3v_1.

Laço é definido como um relacionamento de um objeto com ele mesmo.

A Figura 8 ilustra um exemplo de um laço do nó v_1 do grafo G(V,E) | V=$\{v_1, v_2\}$ e E=$\{v_1v_2, v_1v_1\}$.

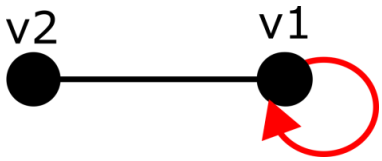

Figura 8 - Laço em um Grafo.

- A *distância* entre dois vértices $v_i \; e \; v_j \in$ V será dada pela quantidade de arestas no caminho entre eles de forma que o comprimento de tal caminho seja o mínimo, ou seja, mesmo que haja mais de um caminho, apenas o de menor comprimento irá representar a distância entre eles. A notação utilizada para expressar tal distância é $d(v,w)$ onde $v = v_i \; e \; w = v_j$. Um exemplo é dado na Figura 7 onde d(v_1,v_6) = 4.

O *grau* de um vértice é dado pela quantidade de arestas que ele possui associadas a ele. Um *grafo regular* é um grafo onde todos os seus vértices possuem o mesmo grau. O grafo K_3 da Figura 9 é um exemplo de um grafo 2-regular, pois todos os seus vértices possuem grau 2.

- Dois vértices são adjacentes se e somente se compartilharem uma mesma aresta entre si, portanto, se existe uma aresta $e_i \in E$ compartilhada por dois vértices $v_i \; e \; v_j \in V$ então ambos podem ser considerados adjacentes. A adjacência de dois vértices pode ser vista como um relacionamento de vizinhança entre eles. No grafo K_3 da Figura 9 os vértices $v_2 \; e \; v_3$ são adjacentes.

Alguns exemplos de classes de grafos podem ser vistos abaixo:

Grafo simples é um grafo não direcionado, sem laços e que não possui mais de uma aresta entre um mesmo par de vértices.

Grafo completo é um grafo G onde cada um de seus vértices está conectado através de uma aresta com todos os demais vértices pertencentes ao grafo. A notação utilizada é K_n onde n representa o número total de vértices do grafo. Um grafo completo possui $\frac{n(n-1)}{2}$ arestas, pois $\forall \, v_i \in V, \; e_i \in E \mid 1 \leq i \leq n,$ deve existir uma aresta e_i conectando-o aos demais $n-1$ vértices do grafo. Isso é resolvido por uma combinação simples $\binom{n}{k}$ onde k=2, pois queremos combinar cada vértice do grafo a todos os demais vértices do grafo, ou seja, formar todos os pares possíveis de forma que a ordem dos elementos não importe ($AB = BA$).

Resolvendo a combinação:

$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)!}{2!(n-2)!} = \frac{n(n-1)(n-2)!}{2!(n-2)!} = \frac{n(n-1)}{2!} = \boldsymbol{\frac{n(n-1)}{2}}$.

A Figura 9 apresenta exemplos de grafos completos.

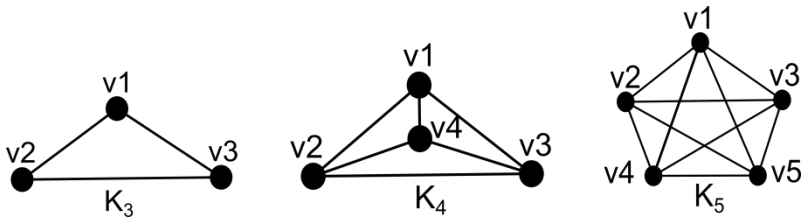

Figura 9 - Grafos Completos K_3, K_4 *e* K_5.

Multigrafo é um grafo onde é permitido múltiplas arestas relacionando dois vértices. A Figura 4 é um exemplo de multigrafo.

- *Grafo conexo* é um grafo onde existe um caminho entre qualquer par de vértices v_i *e* $v_j \in V$, ou seja, todos os pares de vértices devem estar ligados por algum caminho. Consequentemente, se houver pelo menos um vértice isolado em um grafo, ele será classificado como desconexo. O grafo da Figura 5 é um Grafo conexo.

Árvore é um grafo acíclico e conexo. A Figura 10 ilustra uma árvore com 15 vértices e 14 arestas. Em uma árvore com |V| = n, temos |E| = $n - 1$, o livro (Bondy & Murty, 1976) apresenta um estudo mais detalhado. Uma árvore pode ser enraizada, no exemplo da Figura 10 a árvore possui o nó V1 como raiz.

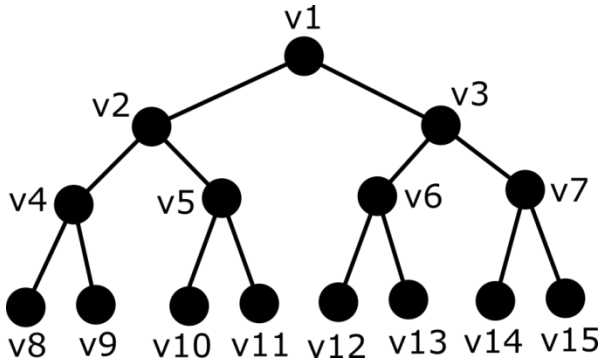

Figura 10 - Exemplo de Árvore.

Subgrafo H(V', E') de um grafo G(V, E) é um grafo onde $V' \subseteq V$ *e* $E' \subseteq E$.
A Figura 11 apresenta um exemplo.

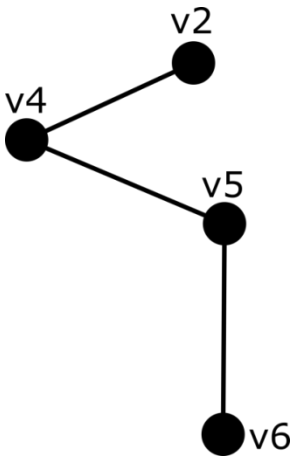

Figura 11 - Exemplo de Subgrafo H(V',E') do grafo da Figura 5.

Subgrafo gerador é dado pela remoção de uma ou mais arestas de um grafo G, mantendo os vértices. O subgrafo resultante é dito gerador de G.

Subgrafo induzido é dado pela remoção de um ou mais vértices preservando as adjacências dos vértices que restam.

Clique é um subgrafo $H(V', E')$ de um grafo $G(V, E)$, onde $H(V', E')$ é um grafo completo.

Grafos ponderados são grafos em que suas arestas possuem pesos, estes pesos podem estar relacionados a várias coisas como, por exemplo, distância, preço, custo, fluxo, confiabilidade, entre outras coisas. O Grafo da
Figura 14 é um exemplo de grafo ponderado.

1.5 COMPLEXIDADE DE ALGORITMOS

1.5.1 DEFINIÇÃO

A complexidade de um algoritmo é um parâmetro para mensurar a eficiência do algoritmo, assunto que pode ser visto com mais detalhes em (Szwarcfiter & Markenzon, 2010). Sua importância se deve ao fato de existirem problemas que, para serem resolvidos requerem algoritmos não triviais e que muitas vezes utilizam de alto processamento e/ou memória. Como recursos de Hardware não são infinitos

e muitas vezes são de alto custo, então é extremamente importante encontrar algoritmos que resolvam o problema utilizando o mínimo de processamento possível.

Os aspectos corretude e eficiência são importantes na análise de algoritmos, análise esta que busca encontrar o melhor algoritmo (solução) dentre muitas para um determinado problema. Esses aspectos podem verificar a viabilidade de implementação prática de algoritmos, ou fazer comparações de eficiência entre eles.

Corretude está diretamente ligada ao resultado da execução do algoritmo em função de seu valor de entrada. Tal resultado, ou valor de saída, deve estar correto, ou seja, deve resolver o problema explorado. Este processo nem sempre é trivial e por diversas vezes utiliza-se de métodos experimentais. Em geral, utilizam-se técnicas matemáticas para testar a corretude de algoritmos.

Eficiência está diretamente ligada ao desempenho do algoritmo seguindo determinado critério. Geralmente é influenciada pelo tamanho e configuração da entrada. Ela é inversa da quantidade de recursos (memória, processamento, armazenamento em disco, etc.) necessários para a execução do algoritmo, ou seja, quanto mais eficiente for um algoritmo menos recursos ele precisará para ser executado. Mais sobre corretude e eficiência de algoritmos pode ser visto em (Carvalho, 2004).

São levadas em consideração as complexidades de pior caso, caso médio e melhor caso durante essa mensuração. Para medir a eficiência de algoritmos são utilizados métodos experimentais e analíticos.

A avaliação da complexidade de um algoritmo busca identificar a quantidade de passos computacionais que serão executados em função do tamanho de sua entrada. A partir da complexidade do algoritmo é possível avaliar o tempo de processamento, comparar a eficiência entre algoritmos e verificar se um algoritmo é o mais eficiente possível, por exemplo.

A complexidade de um algoritmo pode ser calculada reduzindo a função que descreve o número de operações realizadas pelo algoritmo. A função é reduzida eliminando-se primeiramente as constantes aditivas e multiplicativas envolvidas na função, em seguida devem ser verificados os termos que possuem crescimento pequeno e lento em relação aos demais termos (termos inferiores) e eliminá-los também. Por fim, deve restar apenas o(s) termo(s) de crescimento rápido e grande em relação aos demais termos (termos superiores), tais termos são usados para representar a complexidade do algoritmo por serem os que irão consumir maior recurso

computacional, ao contrário dos demais termos que consomem uma parte muito pequena e muitas vezes insignificante.

É importante ressaltar que o tempo de execução do algoritmo avaliado não é mensurado em uma máquina específica, ou seja, é um tempo de execução baseado nos números de operações (passos) que ele realizará. Este tempo pode ser, por exemplo, logarítmico, polinomial ou exponencial dependendo de sua complexidade.

1.5.2 TIPOS DE COMPLEXIDADE

Existem três formas de se medir a complexidade de tempo ou espaço a partir de diferentes entradas de mesmo tamanho, todas elas estão diretamente ligadas a cada entrada em particular, são elas a complexidade de pior caso, melhor caso e caso médio.

Na complexidade de pior caso é considerado o maior número de operações para qualquer que seja a entrada do algoritmo (o alto número de operações implica em alto consumo de recursos como processamento e memória). Essa maneira é considerada "Pessimista", pois ela irá considerar o maior número de operações para qualquer entrada do algoritmo, no entanto, nem todas as entradas usam uma quantidade alta de operações do algoritmo, algumas utilizam o menor número de operações do algoritmo e, portanto, essa forma de medida não é muito precisa. Neste ponto se encaixa a definição de complexidade de melhor caso, que considera um baixo número de operações para qualquer entrada do algoritmo, essa complexidade é considerada "otimista", pois considera que para qualquer entrada possível, o algoritmo irá executar o mínimo de operações e consequentemente terá um baixo consumo de recursos, entretanto, podem existir entradas que utilizam uma quantidade alta de operações, sendo assim essa medida pode também não ser muito precisa.

Buscando uma precisão maior, uma técnica mais apurada e melhor para diferenciar o algoritmo mais eficiente na prática entre algoritmos equivalentes em complexidade é utilizada a complexidade de caso médio. A complexidade de caso médio faz uso da média de todas as entradas possíveis do algoritmo e uma distribuição de probabilidade é aplicada às entradas do algoritmo, sendo assim, serão consideradas todos os tipos de entradas, em particular as "piores" (mais custosas de

recursos) e as "melhores" (menos custosas de recursos) entradas associadas as suas respectivas probabilidades de ocorrência, distribuídas pela função de distribuição de probabilidade utilizada, resultando em um número de operações consideradas pela execução do algoritmo, podendo ser ou não ser alta. Para a complexidade de caso médio o algoritmo irá executar um número de passos igual à complexidade calculada. Visto que nem sempre é fácil obter as informações de todos os tipos de entrada, obter a complexidade média de um algoritmo se torna inviável e muitas situações. Assim sendo, em geral, analisamos os algoritmos pela sua complexidade de pior caso.

Algo importante de se ressaltar é que existe um limite superior e inferior (matematicamente) para cada problema existente, isto é, cada problema possui um número máximo e mínimo de operações (passos) no qual o mínimo de operações representará o limite inferior do problema e o máximo de operações representará o limite superior do problema. Se algum algoritmo resolve um problema com um número de operações igual ao limite inferior do problema, então ele é dito *ótimo*. Para provar um limite superior para um problema estabelecido é suficiente mostrar um algoritmo que tenha custo máximo igual ao limite superior proposto para resolver tal problema, tarefa que muitas das vezes não é fácil. Da mesma forma, provar um limite inferior para determinado problema não é tarefa simples, pois é necessário provar que não existe nenhum algoritmo que resolva o problema com custo inferior ao limite proposto.

1.5.3 NOTAÇÕES

Uma família de notações é utilizada para resumir funções, eliminando termos de crescimento pequeno e lento da lei da função. Para representar limites superiores de funções utilizamos a notação O, para limites inferiores Ω, e para limites justos θ. Utilizamos então essas notações para expressar as complexidades dos algoritmos.

1.5.4 EXEMPLOS

Exemplo 2: Cálculo de complexidade $O(f),\ \Theta(f),\ \Omega(f)$:

$f = n^8 + 2^{\sqrt{n}} = O(2^{\sqrt{n}}), O\left(2^{\sqrt{n \log n}}\right), \Omega(n^8),\ \Omega(n^2), \Theta(2^{\sqrt{n}})$.

$f = n^4 + n^3 \log n = O(n^4), O(n^7), \Omega(n^3 \log n), \Omega(n), \Theta(n^4)$.

$f = 500 = O(1),\ \Omega(1), \Theta(1)$.

$f = \frac{1}{\sqrt{n}} + n = O(n), O(n^3), \Omega\left(\frac{1}{\sqrt{n}}\right), \Omega(1),\ \Theta(n)$.

$f = 1^n + n^2 = O(n^2), O(n^5), \Omega(1^n), \Omega(1),\ \Theta(n^2)$.

Exemplo 3: Alguns algoritmos com suas complexidades:

Obs.: Para cada algoritmo de busca ou de ordenação citados abaixo, n é o tamanho da lista de entrada do algoritmo.

Algoritmo de Busca Binária que possui complexidade de pior caso $O(log_2 n)$.

Algoritmo de ordenação Bubble Sort que possui complexidade de pior caso $O(n^2)$.

Algoritmo de ordenação Merge Sort que possui complexidade de pior caso $O(nlogn)$.

Algoritmo de busca linear que possui complexidade de pior caso $O(n)$.

Algoritmo usual de soma de matrizes tem complexidade $O(n^2)$. Esse algoritmo é ótimo para esse problema pela seguinte razão:

Será necessário acessar cada elemento das matrizes exatamente uma vez, assim sendo o preenchimento de cada elemento da matriz resultante terá complexidade $O(1)$. A complexidade para acessar todos os elementos da matriz é $O(n^2)$ onde $n * m$ é o tamanho da matriz de entrada. Portanto, para cada elemento de uma matriz será necessário percorrer elementos da outra matriz, com isso, o limite inferior desse problema é $O(nn) = O(n^2)$. Visto que o algoritmo de soma de matriz resolve esse problema com complexidade de pior caso $O(n^2)$, ele é dito ótimo para esse problema.

Algoritmo usual de multiplicação de matrizes tem complexidade $O(n^3)$, pois requer aproximadamente n^3operações aritméticas envolvendo adições e multiplicações, fazendo a redução da função temos $O(n^3)$ que representa sua complexidade de pior caso.

Algoritmo de Coppersmith-Winograd para multiplicação de matriz tem complexidade $O(n^{2.375})$, ou seja, sua complexidade é menor que a do algoritmo usual de multiplicação de matrizes. Ele não é ótimo já que no momento de escrita desse trabalho, o limite inferior para esse problema de multiplicação de matrizes era $O(n^2)$ pela mesma razão explicada anteriormente para o algoritmo usual de soma de matrizes. Esse algoritmo também não é ótimo em razão de que existe outro algoritmo que resolve esse problema com complexidade menor que $O(n^{2.375})$.

Um estudo mais completo e detalhado pode ser visto em (Szwarcfiter & Markenzon, 2010).

2 ALGORITMOS EM GRAFOS

Dentre as técnicas existentes para a solução de problemas algorítmicos em grafos, a busca ocupa lugar de destaque pelo grande número de problemas que podem ser resolvidos através de sua utilização. Provavelmente, a importância dessa técnica é ainda maior quando o universo das aplicações for restrito aos algoritmos considerados eficientes. A busca visa resolver um problema básico como o de explorar um grafo, ou seja, dado um grafo deseja-se obter um processo sistemático de como caminhar pelos vértices e arestas do mesmo. Um estudo mais completo e detalhado sobre algoritmos, incluindo os algoritmos estudados neste trabalho, pode ser visto de forma teórica e prática em (Cormen, Charles, Ronald, & Clifford, 2002).

2.1 ALGORITMO DE BUSCA EM PROFUNDIDADE (DEPTH-FIRST SEARCH)

O algoritmo de busca em profundidade tem um funcionamento que pode ser descrito da seguinte forma, ele percorre todos os ramos de um ou mais vértices origem em uma árvore de profundidade antes de retroceder. O algoritmo DFS pode ser

aplicado a diversos grafos, porém, neste trabalho vamos utilizá-lo apenas sobre o grafo do tipo árvore de profundidade. Existem duas formas de implementar este algoritmo, de forma recursiva ou não. O algoritmo pode utilizar a estrutura de dados do tipo FILO (First-in, Last-out) chamado de *pilha* para guardar os nós a serem visitados. A ideia do algoritmo na forma não recursiva é a de escolher um nó que será o nó origem, inseri-lo na pilha e marcá-lo como visitado, em seguida verifica-se se esse nó possui filhos não visitados, caso possua, eles devem ser inseridos na pilha e o passo anterior deve ser feito novamente.

Algoritmo 1:
A execução do algoritmo de forma não recursiva é dada da seguinte forma:

1. Cria-se uma pilha, adiciona-se o nó origem a pilha e marca-o como visitado.
2. Enquanto a pilha não está vazia, faça:
    2.1. Recupere um nó da pilha.
    2.2. Recupere um filho deste nó.
    2.3. Se esse nó filho ainda não foi visitado, faça:
        2.3.1. Marque-o como visitado.
        2.3.2. Insira-o na pilha.
    2.4. Se esse nó filho já foi visitado, faça:
        2.4.1. Remova o nó pai da pilha

A execução do algoritmo de forma recursiva é dada da seguinte forma:

1. Criam-se dois procedimentos, como por exemplo ‘AlgoritmoDFS’ e ‘VisitaNo’ onde o procedimento ‘AlgoritmoDFS’ irá receber um grafo G qualquer como parâmetro, e o procedimento ‘VisitaNo’ irá receber um vértice qualquer deste grafo.

2. O procedimento ‘AlgoritmoDFS’ irá verificar cada vértice do grafo se o mesmo já foi visitado, caso ainda não tenha sido visitado, então o segundo procedimento deverá ser chamado utilizando este vértice como parâmetro.

No Exemplo 4 vemos a execução do Algoritmo 1 DFS não recursivo em relação ao grafo da Figura 5.

Exemplo 4: Seja G o grafo da Figura 5, considerando o vértice v1 como origem e $d(v_1, v_1) = 1$, N' como a ordem em que os vértices foram visitados e $d_i p_i$ representando cada vértice do grafo onde $d_i = d(v_1, v_i)$ e p_i= predecessor de i, sendo $i \in \{v_1, v_2, v_3, v_4, v_5, v_6\}$.

Desta forma a Tabela 1 e a Figura 12 representam a execução do algoritmo como descrito anteriormente. A Figura 12 contém uma tabela hash onde cada entrada representa um vértice do grafo e para cada entrada existe uma lista encadeada associada que representa o relacionamento do vértice com seu vizinho.

Figura 12 - Tabela hash DFS do Grafo da Figura 5.

Passos:	N'	$d_{v_1} p_{v_1}$	$d_{v_2} p_{v_2}$	$d_{v_3} p_{v_3}$	$d_{v_4} p_{v_4}$	$d_{v_5} p_{v_5}$	$d_{v_6} p_{v_6}$
1	v_1	**1**,∞	∞	∞	∞	∞	∞
2	$v_1 v_2$		**2**, v_1	∞	∞	∞	∞
3	$v_1 v_2 v_3$			**3**, v_2	∞	∞	∞
4	$v_1 v_2 v_3 v_4$				**3**, v_2	∞	∞
5	$v_1 v_2 v_3 v_4 v_5$					**4**, v_4	∞
6	$v_1 v_2 v_3 v_4 v_5 v_6$						**5**, v_5

Tabela 1 - Exemplo de execução do algoritmo DFS para o Grafo da Figura 5.

2.1.1 ANÁLISE DE COMPLEXIDADE

Podemos analisar a complexidade do algoritmo recursivo da seguinte forma, no procedimento chamado de 'AlgoritmoDFS', cada vértice do grafo é referenciado

no máximo uma vez e passado como parâmetro na chamada do procedimento ‘VisitaNo’, portanto pode ser agregado a complexidade O (|V|) = O(n).

No procedimento ‘VisitaNo’, cada vértice será marcado como visitado em seu início e logo após, cada um de seus vizinhos será referenciado no máximo uma vez. Considerando que esta referência indica relacionamento, ou seja, as arestas do grafo, podemos então agregar complexidade O (|E|) = O(m). Sendo assim, a complexidade do algoritmo é dada por O (|E| + |V|) = O(n + m).

2.2 ALGORITMO DE BUSCA EM LARGURA (BREADTH-FIRST SEARCH)

O algoritmo de busca em largura pode ser descrito da seguinte forma: O algoritmo fará uma busca em um grafo a partir de um nó origem, percorrendo todos os níveis do grafo, ou seja, após visitar todos os nós em um nível K, ele irá visitar os nós no nível K+1 até que não haja mais níveis para serem visitados, como por exemplo, atingindo um nó folha. O algoritmo utiliza uma estrutura de dados do tipo FIFO (First-in, First-out) chamada fila para armazenar os nós descobertos que posteriormente serão removidos da fila e usados para se descobrir seus vizinhos. Pode-se utilizar vetores para guardar distância entre os nós e seus predecessores. Uma introdução sobre a aplicação do BFS em grafos pode ser vista no artigo (bijulsoni, 2013). Listamos o funcionamento desse algoritmo da seguinte forma.

Algoritmo 2:

1. Cria-se uma fila, adiciona o nó origem a fila e marca-o como visitado.
2. Enquanto a fila não estiver vazia, faça:
    - 2.1. Remova um nó da fila
    - 2.2. Enquanto este nó removido tiver filhos não visitados faça:
        - 2.2.1. Recupere esses nós filhos.
        - 2.2.2. Marque-os como visitados.
        - 2.2.3. Adicione-os a fila.

No Exemplo 5 vemos a execução do Algoritmo 2 BFS no grafo da Figura 5.

Exemplo 5: Seja G o grafo da Figura 5, considerando o vértice v1 como origem e $d(v_1, v_1) = 1$, N' como a ordem em que os vértices foram visitados e $d_i p_i$ representando cada vértice do grafo onde $d_i = d(v_1, v_i)$ e p_i= predecessor de i, sendo $i \in \{v_1, v_2, v_3, v_4, v_5, v_6\}$.

Desta forma a Tabela 2 e a Figura 13 representam a execução do algoritmo como descrito anteriormente. A Figura 13 contém uma tabela hash onde cada entrada representa um vértice do grafo e para cada entrada existe uma lista encadeada associada que representa o relacionamento do vértice com seu vizinho.

Figura 13 - Tabela hash BFS do Grafo da Figura 5.

Passos:	N'	$d_{v_1}p_{v_1}$	$d_{v_2}p_{v_2}$	$d_{v_3}p_{v_3}$	$d_{v_4}p_{v_4}$	$d_{v_5}p_{v_5}$	$d_{v_6}p_{v_6}$
1	v_1	**1**,∞	∞	∞	∞	∞	∞
2	$v_1v_2v_3$		**2**, v_1	**2**, v_1	∞	∞	∞
3	$v_1v_2v_3v_4$				**3**, v_2	∞	∞
4	$v_1v_2v_3v_4v_5$					**4**, v_4	∞
5	$v_1v_2v_3v_4v_5v_6$						**5**, v_5

Tabela 2 - Exemplo de execução do algoritmo BFS para o Grafo da Figura 5.

2.2.1 ANÁLISE DE COMPLEXIDADE

Considerando um grafo G = (V, E), e utilizando os conceitos de complexidade de algoritmos vistos anteriormente, é possível calcular a complexidade deste algoritmo. Cada vértice será colocado na fila no máximo uma vez, e, portanto, será retirado da fila no máximo uma vez. Estas operações de enfileirar e desenfileirar tem complexidade de O (1), e, portanto, o tempo total de operações de filas é O (|V|). Como a lista de adjacências de cada vértice é examinada somente quando o vértice é desenfileirado, a lista de adjacência de cada vértice será então examinada no máximo uma vez.

Considerando que a soma dos comprimentos de todas as listas de adjacências é $\theta(|E|)$, a varredura total das listas de adjacência terá o tempo máximo gasto de O(|E|). A sobrecarga correspondente à inicialização é O (|V|) e, portanto, o tempo de execução total do BFS é O (|V|+|E|). Sendo assim, a busca é executada em tempo linear no tamanho da representação de lista de adjacências de G.

2.3 ALGORITMO DE DIJKSTRA

Dado um grafo G onde cada aresta possui um "peso" (no escopo deste trabalho o “peso” representa a distância real entre um vértice e outro ou uma proporção da distância real), é escolhido um vértice raiz da busca (ponto origem), o algoritmo de Dijkstra calculará então o caminho de custo mínimo do vértice origem a cada um dos demais vértices do grafo G. Ele é bastante simples e com um bom nível de desempenho. Ele não garante, contudo, a exatidão da solução caso haja a presença de arcos com valores negativos. Podemos ver os possíveis passos de execução do algoritmo abaixo:

1. O algoritmo recebe como entrada o grafo, o peso das arestas e um nó origem.

2. Cria-se um vetor N qualquer para armazenar os nós que já possuem o menor caminho definido de acordo com o cálculo da distância dado os custos de cada enlace.

3. A inicialização será feita com o nó de origem sendo incluído no vetor N, e a distância de todos os seus vizinhos até ele sendo calculadas e incluídas em alguma tabela T. A distância dos nós que ainda não foram descobertos deverá ser infinita.

4. Enquanto houverem nós a serem incluídos em nosso vetor, faremos o seguinte:

4.1. Escolheremos o nó V qualquer que possua a menor distância até a origem e que ainda não tenha sido inserido no vetor.

4.2. A distância de cada um dos vizinhos de V em relação ao nó origem será atualizada na tabela T com a soma da distância do nó V até a origem mais a distância de V e seu vizinho somente se está distância for menor do que a existente na tabela T. O nó predecessor também poderá ser guardado na tabela T.

Ao final da execução do algoritmo teremos uma árvore de caminhos mínimos a partir do nó origem, onde o vetor N terá os vértices na ordem de construção da árvore e cada entrada da tabela T correspondente aos vértices do grafo onde terão os seus respectivos custos até a origem e seu nó predecessor.

Descreveremos a seguir um exemplo de utilização do algoritmo de Dijkstra para encontrar um caminho mais curto em uma rede de roteadores.

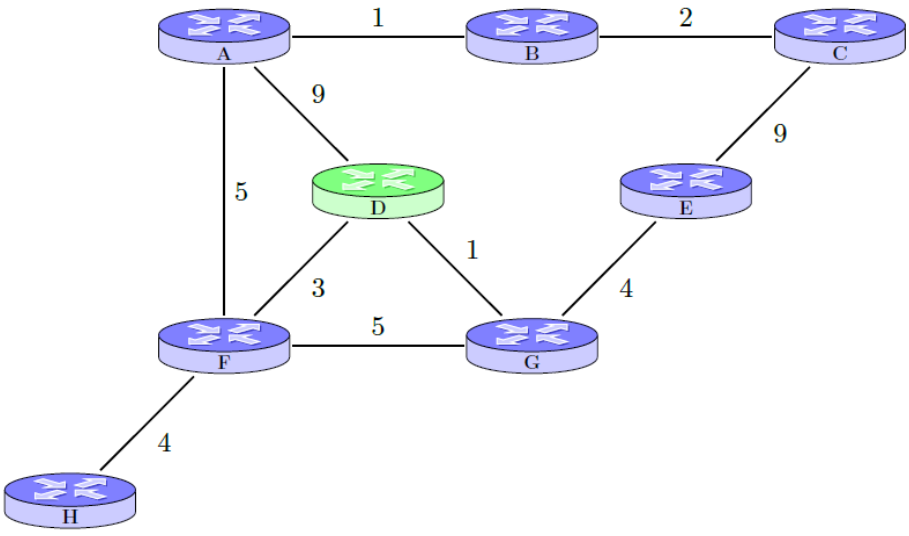

Figura 14 - Rede de roteadores com o roteador D sendo a origem.

Considerando o roteador D como origem, os números nas arestas representando seus respectivos pesos, N' como o vetor que irá conter os roteadores que já possuem o menor caminho definido e $d_i p_i$ representando cada roteador da rede onde $i \in \{A, B, C, D, E, F, G, H\}$, teremos a seguinte tabela abaixo representando a execução do algoritmo como descrito anteriormente:

Passos:	N'	$d_a p_a$	$d_b p_b$	$d_c p_c$	$d_e p_e$	$d_f p_f$	$d_g p_g$	$d_h p_h$
1	D	9,D	Infinito	Infinito	Infinito	**3,D**	**1,D**	Infinito
2	DG	9,D	Infinito	Infinito	5,G			Infinito
3	DGF	8,F	Infinito	Infinito	**5,G**			7,F
4	DGFE	**8,F**	Infinito	14,E				**7,F**
5	DGFEHA		**9,A**	14,E				
6	DGFEHABC			**11,B**				

Tabela 3 - Tabela de execução do algoritmo de Dijkstra com nó origem D.

2.3.1 ANÁLISE DE COMPLEXIDADE

Em uma implementação usando um vetor para armazenar os vértices, cada vértice será armazenado no vetor, agregando a complexidade O (|V|) e para cada operação de extração do vetor o custo será de O (|V|). As operações de visita a vértices vizinhos custarão O (|E|). A complexidade será, portanto $O(|V^2| + |E|) = O(|V^2|)$.

2.4 ÁRVORES GERADORAS MINIMAS (MINIMAL SPANNING TREE)

Se um vértice v da árvore T possuir grau ≤ 1 então v é uma folha. Caso contrário, v é um vértice interior. Toda árvore T com n vértices possui exatamente n – 1 arestas. Um grafo G é uma árvore se e somente se existir um único caminho entre cada par de vértices de G.

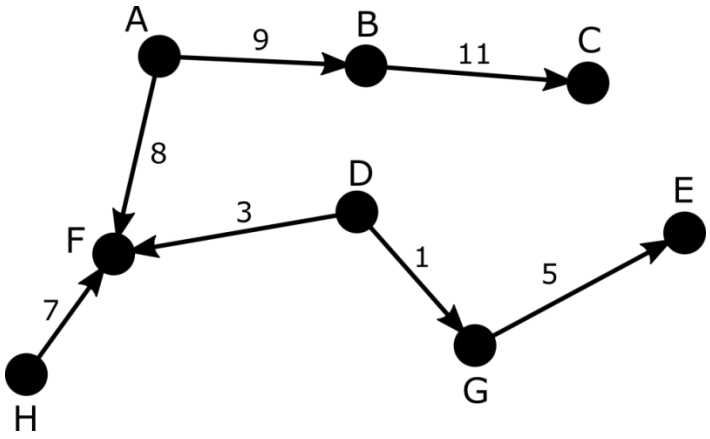

Figura 15 - Exemplo de MST para o grafo da seção Algoritmo De Dijkstra.

Uma árvore geradora é uma árvore T, subgrafo de G, que contém todos os nós de G. Uma árvore geradora cuja a soma dos pesos de seus arcos seja menor do que em qualquer outra situação é chamada de árvore geradora mínima. A Figura 15 ilustra uma árvore geradora mínima.

Um grafo com G com n nós, possui N^{N-2} árvores geradoras diferentes sendo algumas delas árvores mínimas possíveis. Serão apresentados dois algoritmos que ajudam a resolver esse problema, o algoritmo de Kruskal e o algoritmo de Prim. Esses dois algoritmos são gulosos, isto é, utilizam de escolhas locais ótimas em cada etapa, buscando ao final ter uma solução global ótima. Mais detalhes podem ser vistos em (Contribuidores, Algoritmo guloso, 2016).

2.4.1 ALGORITMO DE KRUSKAL

O algoritmo de Kruskal tem o seu funcionamento definido da seguinte forma: Dado uma árvore, façamos o seguinte até encontrar uma árvore geradora mínima:

- Escolher, do conjunto de arestas E, aquela que possua o menor peso.
- Se a inclusão desta aresta na solução não formar um ciclo, incluímos a aresta.

- Caso contrário, descartamos a aresta e a retiramos de E.

2.4.1.1 ANÁLISE DE COMPLEXIDADE

A complexidade do algoritmo de Kruskal é dada da seguinte forma. Em um grafo com n nós e m arestas, o número de operações é:

1. O (m log m), para ordenar as arestas, que é equivalente à O (m log n);
2. O (n) para inicializar os conjuntos distintos de cada componente conexa.
3. No pior caso, $O\left(\left(2m + \left(n - 1\right)\right) * \log n\right)$ para determinar e misturar as componentes conexas.
4. O (m) para o restante das operações.
5. Conclui-se que o tempo total para o algoritmo de Kruskal é O (m log n).

2.4.2 ALGORITMO DE PRIM

O algoritmo de Prim funciona de maneira bastante semelhante ao anterior. Iniciamos com uma árvore T (V,E) formada por um único nó n (qualquer nó do grafo) e vamos adicionando à árvore, a cada passo, um nó m que estiver mais próximo de n. Poderíamos resumir o algoritmo assim: Dado um grafo formado pelo conjunto de nós N e o conjunto de arestas E, escolha um nó qualquer do grafo e coloque na árvore T. Repita os seguintes passos até que T seja uma árvore geradora:

- Escolher o nó que está mais próximo da árvore T, ou seja, o nó que está ligado a um nó de T pela aresta de menor valor.
- Se a inclusão deste nó não fecha um ciclo, faça:
- Adicionar este nó a T. (Paramos de adicionar quando todos os nós do grafo forem adicionados).

2.4.2.1 ANÁLISE DE COMPLEXIDADE

O loop principal do algoritmo de Prim é executado n − 1 vezes e em cada iteração, o laço interno tem uma complexidade O (n). Portanto, o algoritmo de Prim tem complexidade $O\ (n^2)$.

3 IMPLEMENTAÇÃO

Para a realização prática deste trabalho de conclusão de curso, foi implementada uma classe em Python a fim de ser importada por qualquer aplicação ou mecanismo (por exemplo um Drone) que possua alguma forma de detectar um obstáculo à sua frente e que conheça sua posição atual geodésica, assim como a posição geodésica destino. Um ponto importante que merece ser ressaltado é a escolha da utilização da linguagem Python para desenvolvimento dessa aplicação, já que a maioria dos aplicativos desenvolvidos hoje para Drones são escritos e interpretados por essa linguagem, o que torna mais simples a importação da classe desenvolvida neste TCC por aplicativos de terceiros.

Essa classe possui métodos que tem como finalidade corrigir um percurso desviando desse obstáculo, que pode ser qualquer objeto inesperado durante o percurso que crie uma barreira entre o Drone e seu destino. Um grafo para a situação anterior pode ser visto abaixo:

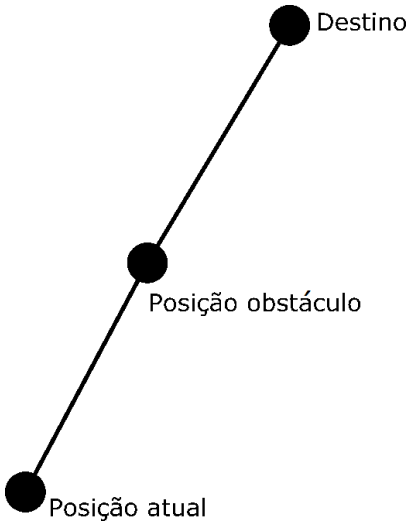

Figura 16 – Grafo representando os três pontos geodésicos do Drone.

As principais etapas que devem ser seguidas a fim de obter a correção do percurso dado, que serão mais detalhadas nas próximas seções, são: o recebimento dos dois pontos conforme o grafo da figura anterior em latitude e longitude (Posição atual e do obstáculo), a conversão dos pontos dados em coordenadas cartesianas e a utilização de cálculos matemáticos para criação de um novo ponto no grafo. Tal ponto será retornado como resposta.

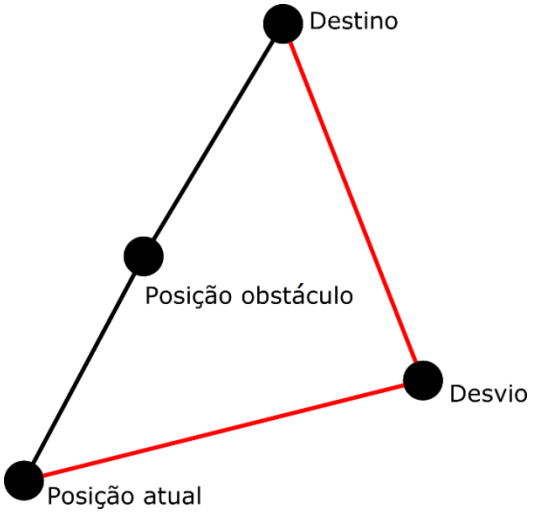

Figura 17 - Grafo representando a adição do ponto de desvio na rota.

3.1 RECEBENDO PARÂMETROS

Como havíamos afirmado anteriormente, o primeiro passo é instanciar uma nova variável do tipo ‘coordinatesGraphs’ e inserir os valores geodésicos do ponto atual e do obstáculo utilizando os métodos ‘addValueCurrent’ e ‘addValueObstacle’, respectivamente, conforme o trecho de código abaixo:

```
t = coordinatesGraphs()
t.addValueCurrent(40, 42, 9.716, 'N', 74, 4, 52.188, 'W', 2)
t.addValueObstacle(40, 42, 10.115, 'N', 74, 4, 52.1446, 'W', 2)
```

Internamente os métodos invocados anteriormente irão guardar as variáveis em propriedades da classe a fim de serem utilizadas posteriormente. Todos os dois métodos de adição de valores possuem implementação muito semelhante, logo escolhemos para exemplificar aqui o método ‘addValueCurrent’. Eis o trecho do código que demonstra o processo explicado anteriormente:

```
def addValueCurrent(self, LatDegrees, LatMinutes, LatSeconds, Hemisphere,
                LongDegress, LongMinutes, LongSeconds, Meridian, Altitude):
                self.__setLatitude(LatDegrees, LatMinutes, LatSeconds, Hem-
            isphere, 0)
                self.__setLongitude(LongDegress,  LongMinutes,  LongSeconds,
            Meridian, 0)
                self.__altitude = Altitude
```

3.2 TRANSFORMANDO AS COORDENADAS GEODÉSICAS EM CARTESIANAS

Após a adição dos dois pontos do grafo, o passo seguinte é invocar o método ‘generateRoute’ que retorna como resposta a latitude e longitude do novo ponto para onde o Drone será desviado.

Outro ponto importante que é interessante esclarecer é que os passos anteriores podem ser utilizados recursivamente, ou seja, caso o Drone encontre outro obstáculo durante o percurso corrigido ou, até mesmo, quando o obstáculo for grande demais e os cálculos realizados não tenham sido suficientes para desviar do mesmo e o sensor do Drone detecte novamente o obstáculo, basta realizar novamente os passos iniciais até obter um percurso ideal.

A fim de utilizar geometria analítica para a inserção do novo ponto no grafo, a primeira etapa a ser realizada pelo método ‘generateRoute’ é transformar os pontos inseridos em Graus, Minutos e Segundos em um valor numérico geodésico decimal. Para tal, utilizamos regras simples de transformação que consistem em somar o valor dos Graus com os Minutos divididos por 60 e com os segundos divididos por 3600. Além disso, caso o hemisfério inserido seja Sul ou o meridiano Oeste devemos multiplicar o valor anterior por -1. Mais detalhes podem ser vistos em (IBGE, Frequently Asked Questions, 2015).

Segue método que realiza tais conversões:

```
    index = 0

    while index <= 1:
      self.__numericalLatitude[index] = self.__latitude[index].Degrees +
(self.__latitude[index].Minutes / 60) + (self.__latitude[index].Seconds /
3600)

      if self.__latitude[index].Hemisphere == 'S':
        self.__numericalLatitude[index] = self.__numericalLatitude[index]
* -1

      self.__numericalLogintude[index] = self.__longitude[index].Degrees +
(self.__longitude[index].Minutes / 60) + (self.__longitude[index].Seconds /
3600)

      if self.__longitude[index].Meridian == 'W':
        self.__numericalLogintude[index] = self.__numericalLogintude[index]
* -1

      index = index + 1;
```

Em seguida, os valores numéricos gerados devem ser convertidos em coordenadas cartesianas. Para isso utilizamos como apoio a documentação disponibilizada pelo IBGE em seu site, a documentação demonstra como realizar tais conversões utilizando o sistema de referência geodésico regional para a América do Sul SAD-69. Que foi substituído pelo SIRGAS 2000 (Sistema de Referência Geocêntrico para as Américas 2000) em 2015.

A principal diferença entre SAD69 e o SIRGAS2000 está em suas definições/orientações, isto é, o SAD69 tem definição/orientação topocêntrica enquanto que o SIRGAS2000 tem definição/orientação geocêntrica com referencial de origem de três eixos cartesianos localizados no centro da Terra. Além disso, as técnicas de posicionamento deles são diferentes, O SAD 69 utiliza técnicas clássicas (triangulação e poligonação), já o SIRGAS2000 utiliza os sistemas globais de navegação (posicionamento) por satélites.

No caso de optar pela conversão entre os sistemas, é possível fazer uso de um sistema intermediário que é equivalente ao SIRGAS 2000 e que na prática não possui diferença significativa, tal sistema é o WGS 84 em sua versão mais atual. Outra opção de conversão é a conversão direta entre SAD-69 e SIRGAS 2000 que pode ser vista em (IBGE, Resolução PR – 1/2005, 2005). A conversão entre SAD-69 e WGS 84, incluindo detalhes da conversão de coordenadas geográficas em coordenadas cartesianas pode ser vista em (IBGE, Resolução PR-22/83, 1989).

Mais detalhes sobre os sistemas de referência utilizados no Brasil podem ser vistos em (IBGE, Frequently Asked Questions, 2015).

Conforme a documentação do IBGE implementamos o método abaixo que recebe três parâmetros, sendo eles a latitude, longitude e a altitude e retorna os pontos cartesianos respectivos x, y e z.

```
  def convertGeodesicsToCartesian(self, geodeticLatitude, geodeticLongitude, ellipsoidalAltitude):

        geodeticLatitude = radians(geodeticLatitude)

        geodeticLongitude = radians(geodeticLongitude)

        ellipsoidalAltitude = radians(ellipsoidalAltitude)
```

```
        flatteningEllipsoid = 1/298.25

        firstEllipsoidEccentricity      =      sqrt(flatteningEllipsoid*(2-
flatteningEllipsoid))

        semiMajorAxis = 6378160

        radiusCurvatureFirstVertical  =  semiMajorAxis/sqrt(1-  firstEllip-
soidEccentricity**2 * sin(geodeticLatitude)**2)

        x    =    (radiusCurvatureFirstVertical    +    ellipsoidalAltitude)*
cos(geodeticLatitude) * cos(geodeticLongitude)
        y    =    (radiusCurvatureFirstVertical    +    ellipsoidalAltitude)*
cos(geodeticLatitude) * sin(geodeticLongitude)

        z = (radiusCurvatureFirstVertical * (1 - firstEllipsoidEccentrici-
ty**2) + ellipsoidalAltitude) * sin(geodeticLatitude)

        return (x,y,z)
```

3.3 GERANDO UM NOVO VERTICE NO GRAFO

A geração do novo ponto no Grafo G utiliza de conceitos da geometria analítica de interseção de segmentos de retas perpendiculares e equação da circunferência. (Gómez, Frensel, & Santo, 2010) Também foi utilizado o conceito da álgebra linear de vetor normal. Os passos a seguir demonstram o passo a passo para a construção do algoritmo para gerar o novo ponto no Grafo G:

1º Passo: Calcular a distância d entre o ponto origem $P(x_1, y_1)$, que é o ponto onde o Drone está posicionado após encontrar o obstáculo, e o ponto $S(x_2, y_2)$, que é o ponto onde foi encontrado o obstáculo.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

2º Passo: Definir e resolver a equação da reta r que passa pelos pontos P e S, seguindo em direção ao ponto Destino no Grafo G. Definir também v, que é o vetor normal à reta r.

$$\begin{cases} y_1 = ax_1 + b \\ y_2 = ax_2 + b \end{cases}$$

$$\therefore b = y_1 - ax_1$$

$$\Rightarrow y_2 = ax_2 + y_1 - ax_1$$

$$\Rightarrow a = \frac{y_2 - y_1}{x_2 - x_1}$$

$$\Rightarrow b = y_1 - \left(\frac{y_2 - y_1}{x_2 - x_1}\right) x_1$$

$$r: y = \left(\frac{y_2 - y_1}{x_2 - x_1}\right) x + y_1 - \left(\frac{y_2 - y_1}{x_2 - x_1}\right) x_1$$

$$r: \left(\frac{y_2 - y_1}{x_2 - x_1}\right) x - y = \left(\frac{y_2 - y_1}{x_2 - x_1}\right) x_1 - y_1$$

$$v = \left(\frac{y_2 - y_1}{x_2 - x_1}, -1\right)$$

3º Passo: Definir a equação da reta s que é perpendicular a r. Aplica-se o ponto S, indicando assim o ponto de interseção entre elas. L é um ponteiro que define S.

$$s: x + \left(\frac{y_2 - y_1}{x_2 - x_1}\right) y = l$$

$$\Rightarrow l = x_2 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right) y_2$$

$$\Rightarrow s: x + \left(\frac{y_2 - y_1}{x_2 - x_1}\right) y = x_2 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right) y_2$$

4º Passo: Obter P'e P'', onde $P'(x', y')$ e $P''(x'', y'')$ são as interseções de s com a circunferência de centro S e raio d.

$$\begin{cases} x + \left(\frac{y_2 - y_1}{x_2 - x_1}\right) y = x_2 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right) y_2 \\ (x - x_2)^2 + (y - y_2)^2 = d^2 \end{cases}$$

$$\begin{cases} x + \left(\frac{y_2 - y_1}{x_2 - x_1}\right) y = x_2 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right) y_2 \\ (x - x_2)^2 + (y - y_2)^2 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}^2 \end{cases}$$

Isolando x na primeira equação e substituindo na segunda equação:

$$\Rightarrow x = x_2 + \frac{y_2 - y_1}{x_2 - x_1} y_2 - \frac{y_2 - y_1}{x_2 - x_1} y$$

$$\Rightarrow \left(x_2 + \frac{y_2 - y_1}{x_2 - x_1} y_2 - \frac{y_2 - y_1}{x_2 - x_1} y - x_2 \right)^2 + (y - y_2)^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

$$\Rightarrow \left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 (y_2 - y)^2 + (y - y_2)^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

Em razão de $(y - y_2)^2 = (-1(y_2 - y))^2 = (-1)^2 (y_2 - y)^2$, a equação é fatorada com o termo $(y_2 - y)^2$ sendo colocado em evidência:

$$(y_2 - y)^2 \left(\left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 + 1 \right) = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

$$\Rightarrow (y_2 - y)^2 = \frac{(x_2 - x_1)^2 + (y_2 - y_1)^2}{\left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 + 1}$$

$$\Rightarrow y_2 - y = \sqrt{\frac{(x_2 - x_1)^2 + (y_2 - y_1)^2}{\left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 + 1}}$$

$$\Rightarrow y = y_2 \pm \sqrt{\frac{(x_2 - x_1)^2 + (y_2 - y_1)^2}{\left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 + 1}}$$

$$\Rightarrow y' = y_2 + \sqrt{\frac{(x_2 - x_1)^2 + (y_2 - y_1)^2}{\left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 + 1}}$$

$$\Rightarrow y'' = y_2 - \sqrt{\frac{(x_2 - x_1)^2 + (y_2 - y_1)^2}{\left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 + 1}}$$

Substituindo y em x:

$$\Rightarrow x = x_2 + \frac{y_2 - y_1}{x_2 - x_1} y_2 - \frac{y_2 - y_1}{x_2 - x_1} \left(y_2 \pm \sqrt{\frac{(x_2 - x_1)^2 + (y_2 - y_1)^2}{\left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 + 1}} \right)$$

$$\Rightarrow x' = x_2 + \frac{y_2 - y_1}{x_2 - x_1} y_2 - \frac{y_2 - y_1}{x_2 - x_1} \left(y_2 + \sqrt{\frac{(x_2 - x_1)^2 + (y_2 - y_1)^2}{\left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 + 1}} \right)$$

$$\Rightarrow x'' = x_2 + \frac{y_2 - y_1}{x_2 - x_1} y_2 - \frac{y_2 - y_1}{x_2 - x_1} \left(y_2 - \sqrt{\frac{(x_2 - x_1)^2 + (y_2 - y_1)^2}{\left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 + 1}} \right)$$

Essa equação tem duas soluções possíveis para y. A escolha do novo ponto para o desvio será uma escolha aleatório dentre P' ou P'' no Grafo G indicando o possível desvio do Obstáculo atual. A figura 18 ilustra o cenário de implementação relativo ao desvio do obstáculo descrito acima.

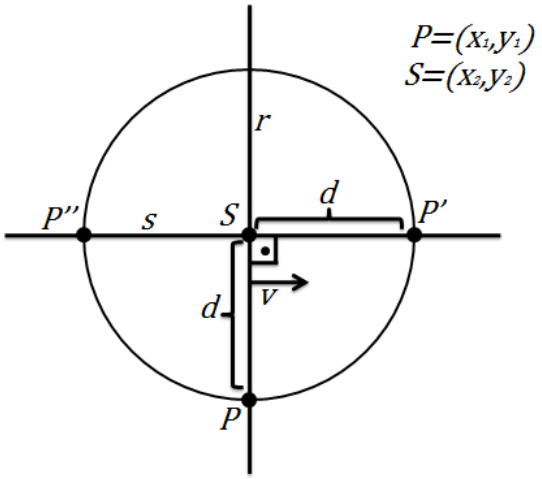

Figura 18 – Cenário de implementação relativo ao desvio do obstáculo.

Conforme explicação anterior, construímos um método chamado CalculateOffset que receberá como parâmetro de entrada duas coordenadas cartesianas e retornará somente uma sendo ela justamente o nosso desvio definido anteriormente.

```
def CalculateOffset(self, x1, y1, x2, y2):
  if (x2 - x1) == 0:
    raise ValueError('Sem solução')
```

```
          if (randint(0,1)) :
            y3 = y2 - sqrt(((x2 - x1)**2 + (y2 - y1)**2)/(((y2 - y1)/(x2 -
x1))**2 + 1))
          else :
            y3 = y2 + sqrt(((x2 - x1)**2 + (y2 - y1)**2)/(((y2 - y1)/(x2 -
x1))**2 + 1))

          x3 = x2 + ((y2 - y1)/(x2 - x1) * y2 - ((y2 - y1)/(x2 - x1)) * y3)

          return (x3,y3)
```

Após obter a resposta do método CalculateOffset é necessário agora apenas converter novamente essa cordenada cartesiana para um ponto geodésico. Para tal, utilizamos como apoio novamente a documentação presente do portal do IBGE e construímos o método intitulado convertCartesianToGeodesics. Segue código fonte:

```
        def convertCartesianToGeodesics(self, x, y, z):

          flatteningEllipsoid = 1/298.25

          firstEllipsoidEccentricity     =     sqrt((flatteningEllipsoid*(2-
flatteningEllipsoid)))

          secondEllipsoidEccentricity                                    =
sqrt(firstEllipsoidEccentricity**2/(1 - (firstEllipsoidEccentricity**2)))

          semiMajorAxis = 6378160

          semiMinorAxis = 6356774.7199

          tan_u     =     (z     /     (sqrt((x**2)     +     (y**2))))     *
(semiMajorAxis/semiMinorAxis)

          cos_u = 1 / (sqrt( 1 + tan_u**2 ))

          sen_u = tan_u / (sqrt( 1 + tan_u**2 ))

          numerator   =   z   +   (secondEllipsoidEccentricity   **   2)   *
semiMinorAxis * (sen_u**3)
```

```
        denominator = sqrt((x**2)+(y**2)) -
(firstEllipsoidEccentricity**2) * semiMajorAxis * (cos_u**3)

        geodeticLatitude = atan(numerator/denominator);

        geodeticLatitude = degrees(geodeticLatitude)

        geodeticLongitude = atan(y/x);

        geodeticLongitude = degrees(geodeticLongitude)

        geodeticLatitudeAux = radians(geodeticLatitude)

        radiusCurvatureFirstVertical = semiMajorAxis/sqrt(1-
firstEllipsoidEccentricity**2 *sin(geodeticLatitudeAux)**2)

        ellipsoidalAltitude = (sqrt((x**2) +
(y**2))/cos(geodeticLatitudeAux)) - radiusCurvatureFirstVertical

        ellipsoidalAltitude = degrees(ellipsoidalAltitude)

        return(geodeticLatitude, geodeticLongitude, ellipsoidalAltitude);
```

Todo passo a passo detalhado anteriormente (com exceção dos métodos de adição de coordenadas) é processado por um método chamado generateRoute que retornará como resposta o desvio calculado.

Uma aplicação de terceiros poderá utilizar qualquer um dos métodos presentes na classe, porém para a completa execução conforme idealizado neste trabalho as seguintes linhas de código devem ser escritas:

```
        t = coordinatesGraphs()
        t.addValueCurrent(self, 22, 54, 14.2, 'S', 43, 07, 48.3, 'W', 1)
        t.addValueObstacle(self, 22, 54, 14.1, 'S', 43, 07, 48.3, 'W', 1)

        answer = t.generateRoute();
```

Importante ressaltar que todos os cálculos realizados são com base no sistema de referência SAD-69. Logo, o software que utilizará esse código deverá

estar de acordo com esse sistema para que não haja divergência na representação dos pontos geodésicos.

O acesso completo as classes com seus respectivos algoritmos encontra-se armazenado em um repositorio Bitbucket que pode ser acessado através do link: https://bitbucket.org/eliaquimauricio/tcc-coordinatesgraphs

CONCLUSÕES E TRABALHOS FUTUROS

Exploramos diversos assuntos nesse trabalho com o objetivo de contribuir para a construção de softwares aplicados a Drones e em robótica no geral. Foi utilizada uma abordagem informal com uma linguagem e exemplos simples, de forma que o entendimento possa ser adquirido mais rápido, porém sem muitos detalhes. Esperamos que tudo que foi descrito neste trabalho possa auxiliar o ensino de algoritmos e em especial algoritmos em grafos com aplicações em problemas do mundo real.

O problema levantado nesse trabalho se refere a algo comum e rotineiro na robótica, que é o desvio de obstáculos de forma autônoma pela máquina. Nossa solução para tal problema utiliza do estudo de grafos para modelagem e entendimento do problema, incluindo o estudo de algoritmos existentes como o DFS, BFS, Dijkstra, Kruskal e Prim com suas respectivas complexidades. Também foram utilizados conceitos clássicos da geometria analítica como, teorema de Pitágoras, equação da reta, equação da circunferência, inserção de segmentos e interseção de segmento e circunferência. A linguagem de programação Python foi usada por ser a linguagem de diversos aplicativos e bibliotecas de Drones e também por ser uma linguagem bem difundida pelo mundo e amplamente usada no ensino de programação de computadores (isso é verdade no momento de desenvolvimento desse trabalho, podendo não ser verdade após algum tempo visto que novas linguagens mais didáticas surgem de tempos em tempos). É importante ressaltar que embora os algoritmos e a implementação tenham sido escritos em linguagens específicas, ambos podem ser escritos e usados em várias linguagens de programação (C, C++, C#, Delphi, Swift, Java, Ruby, entre outras).

Subimos o primeiro degrau de uma longa escada nessa pesquisa, contudo, ainda há muito que ser analisado e otimizado como, por exemplo, a latência na comunicação do software com o controlador da máquina, outros sensores (sonares, radares, câmeras e outros) e tipos de entradas (coordenadas, distância do obstáculo, ângulo de abertura, direção geográfica, entre outros) para o algoritmo.

REFERÊNCIAS BIBLIOGRÁFICAS

- NetworkX developer team. (1 de Maio de 2016). *NetworkX Home*. Acesso em 2 de Novembro de 2016, disponível em NetworkX: http://networkx.github.io/#
- bijulsoni. (1 de Janeiro de 2013). *Introduction to Graph with Breadth First Search(BFS) and Depth First Search(DFS) Traversal Implemented in JAVA*. Acesso em 5 de Outubro de 2016, disponível em codeproject: https://www.codeproject.com/articles/32212/introduction-to-graph-with-breadh-first-search-bf
- Bondy, J. A., & Murty, U. S. (1976). *Graph theory with applications* (Vol. 290). London, Inglaterra: Macmillan.
- C. d. (18 de Novembro de 2016). *Python*, 47244756. Acesso em 20 de Novembro de 2016, disponível em Wikipédia, a enciclopédia livre: https://pt.wikipedia.org/w/index.php?title=Python&oldid=47244756
- Carvalho, M. A. (Fevereiro de 2004). *Análise de algoritmos.* Acesso em 02 de Dezembro de 2016, disponível em Unicamp: http://www.ft.unicamp.br/~magic/analisealgo/apoalgoritmos_ceset_magic.pdf
- Contribuidores, W. (22 de Novembro de 2016). *Algoritmo guloso*. Acesso em 26 de Novembro de 2016, disponível em Wikipédia, a enciclopédia livre: https://pt.wikipedia.org/w/index.php?title=Algoritmo_guloso&oldid=47278373
- Contribuidores, W. (24 de Outubro de 2016). *Veículo aéreo não tripulado*. Acesso em 2016 de Novembro de 01, disponível em Wikipédia, a enciclopédia livre: https://pt.wikipedia.org/w/index.php?title=Ve%C3%ADculo_a%C3%A9reo_n%C3%A3o_tripulado&oldid=47035208
- Contributors, 3D Robotics. (2015). *DroneKit*. Acesso em 05 de setembro de 2016, disponível em DroneKit: http://dronekit.io/
- Cormen, T. H., C. E., R. L., & Clifford. (2002). *Algoritmos: Teoria e Prática* (Vol. 2). Rio de Janeiro: Editora Campus.
- Dahis, R. (2009). *Redes Complexas*. Acesso em 19 de Novembro de 2016, disponível em Redes Complexas: http://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2009_2/dahis/intro.html
- F. D. (2011). Introdução a Redes Complexas. (A. F. Jr, Ed.) *Atualizaçoes em Informática* , 303--358.

- Gómez, J. J., Frensel, K. R., & Santo, N. d. (2010). *Geometria Analítica I* (3° ed.). Rio de Janeiro: Tereza Queiroz.
- *Graph-tool performance comparison*. (27 de outubro de 2015). Acesso em 5 de Novembro de 2016, disponível em Graph-tool: https://graph-tool.skewed.de/performance
- IBGE. (11 de Fevereiro de 2015). *Frequently Asked Questions*. Acesso em 26 de Novembro de 2016, disponível em IBGE: http://www.ibge.gov.br/home/geociencias/geodesia/pmrg/faq.shtm#1
- IBGE. (25 de Fevereiro de 2005). *Resolução PR – 1/2005*. Acesso em 14 de Novembro de 2016, disponível em IBGE: ftp://geoftp.ibge.gov.br/metodos_e_outros_documentos_de_referencia/normas/rpr_01_25fev2005.pdf
- IBGE. (21 de Fevereiro de 1989). *Resolução PR-22/83*. Acesso em 17 de Novembro de 2016, disponível em IBGE: ftp://geoftp.ibge.gov.br/metodos_e_outros_documentos_de_referencia//normas/rpr_2389.pdf
- J. K., & É. T. (2005). *Algorithm Design.* Addison-Wesley.
- Python Software Foundation. (16 de Dezembro de 2016). *Python documentation*. Acesso em 17 de Dezembro de 2016, disponível em Docs Python: https://docs.python.org/
- Szwarcfiter, J. L., & Markenzon, L. (2010). *Estruturas de dados e seus algoritmos.* Rio de Janeiro: LTC.
- Tanner, S., & Brueck, D. (2001). *Python 2.1 Bible.* New York: Hungry Minds.
- *The Boost Graph Library*. (17 de agosto de 2003). Acesso em 2 de Novembro de 2016, disponível em Boost C++ Libraries: http://www.boost.org/doc/libs/1_62_0/libs/graph/doc/index.html
- The igraph core team. (2015). *igraph*. Acesso em 2 de Novembro de 2016, disponível em igraph: http://igraph.org/
- W. C. (06 de Novembro de 2016). *Dynamic programming language*. Acesso em 12 de Novembro de 2016, disponível em Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=Dynamic_programming_language&oldid=748160587

- W. C. (28 de Outubro de 2016). *Functional programming*. Acesso em 02 de Novembro de 2016, disponível em Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=Functional_programming&oldid=746633963
- W. C. (24 de Novembro de 2016). *Graph theory*. Acesso em 30 de Novembro de 2016, disponível em Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=Graph_theory&oldid=751253810
- W. C. (10 de Novembro de 2016). *High-level programming language*. Acesso em 12 de Novembro de 2016, disponível em Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=High-level_programming_language&oldid=748858739
- W. C. (09 de Novembro de 2016). *Imperative programming*. Acesso em 12 de Novembro de 2016, disponível em Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=Imperative_programming&oldid=748664038
- W. C. (09 de Novembro de 2016). *List of programming languages by type*. Acesso em 12 de Novembro de 2016, disponível em Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=List_of_programming_languages_by_type&oldid=748662765
- W. C. (2 de Novembro de 2016). *Metaprogramming*. Acesso em 2 de Novembro de 2016, disponível em Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=Metaprogramming&oldid=749155124
- W. C. (30 de Outubro de 2016). *OpenMP*. Acesso em 2 de Novembro de 2016, disponível em Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=OpenMP&oldid=747543845
- W. C. (02 de Novembro de 2016). *Programming paradigm*. Acesso em 12 de Novembro de 2016, disponível em Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/w/index.php?title=Programming_paradigm&oldid=747479954
- W. C. (08 de Novembro de 2016). *Strong and weak typing*. Acesso em 12 de Novembro de 2016, disponível em Wikipedia, The Free Encyclopedia:

https://en.wikipedia.org/w/index.php?title=Strong_and_weak_typing&oldid=748419394

- *What is graph-tool?* (s.d.). Acesso em 2 de Novembro de 2016, disponível em graph-tool: https://graph-tool.skewed.de/