# Documentação Trabalho Prático 01 de Algorítimos e Estruturas de Dados III

Paulo Cirino Ribeiro Neto
Engenharia de Sistemas
UFMG

23 de outubro de 2015

## 1 Introdução

O objetivo desse trabalho é a resolução do problema do bombeiro preguiçoso que pode ser definido como encontrar o caminho com menor probabilidade de incêndio que permaneça a uma distância máxima predefina de alguma estação do corpo de bombeiros para que ele possa sair de um ponto inicial e se mover até um ponto final.

Mesmo esta definição é de um problema lúdico e que não exite na realidade, é fácil imaginar problemas semelhantes que sejam reais e de grande importância. Um Exemplo ainda utilizando os bombeiros seria definir um caminho com probabilidade máxima de encontrar incêndios e que permaneça perto de hidrantes para ser reabastecido.

Em suma esse trabalho têm o objetivo de fazer com que nós, os alunos, utilizássemos a teoria de grafos, apresentada no segundo modulo da disciplina , em um problema de programação.

## 2 Solução do Problema

### 2.1 Modelagel do Problema

Esse trabalho foi modelado a partir da teoria dos grafos. Nossa modelagem foi definida de forma que os quarteirões são os vértices e as ruas são as arestas.

Além disso utilizamos 2 algorítimos apresentados em sala para resolver o problema, o algorítimo de Busca em Largura(BFS) para definir quais quarteirões estão dentro da distância máxima de um corpo de bombeiros permitida, e o algorítimo de Dijkstra para encontrar o caminho com menor probabilidade de incêndio.

Um dos parâmetros que tivemos que modelar é a função custo das arestas. O motivo disso é que a probabilidade de incêndio entre o caminho 0-3-4, $P(0,3,4) \neq P(0,3) + P(3,4)$. Para resolvermos esse problema fazemos uma

operações com cada probabilidade. A primeira é calcular a probabilidade de não passar em um incêndio no caminho que é $P_N(0,3,4) = 1 - P(0,3,4) = 1 - P(0,3) * P(0,4)$.

Infelizmente isso ainda nos deixa com um grande problema, pois a função custo no algorítimo de Dijkstra têm de ser algo do como $P(0,3,4) = P(0,3) + P(0,4)$, porque no próprio algorítimo o custo até um vértice é o custo até seu anterior mais o custo entre os dois. Para modelar o custo então da forma que funciona para o algorítimo utilizamos então a propriedade da função logarítmica, $\log a * \log b = \log a + \log b$ .

Isso nos permite então definir nossa função custo de cada aresta como $C(x,y) = -\log(1 - P(x,y))$.

## 2.2 Pseudo-Código

### 2.2.1 Programa Principal

Abaixo iremos demonstrar um pseudo-código de como o programa principal funciona, as funções e métodos que estão presentes no mesmo não representam necessariamente funções do programa, mas sim a lógica geral de cada parte do mesmo.

**Result**: Caminho com menor probabilidade de insendio dentro dos vertices permitidos
Parametros = le.parametros();
G = initGrafo(Parametros);
**for** *cont=1; cont* $\leq$ *numQuarteirões; cont++* **do**
  add.Aresta(G, le.aresta() );
**end**
**for** *cont=1; cont* $\leq$ *numBombeiros; cont++* **do**
  marcaPermitidoDistMaxima(G , le.bombeiro());
**end**
Caminho = Dijkstra(G);
print.Caminho();

**Algorithm 1:** Pseudo-código Main()

### 2.2.2 add.Aresta()

Agora segue abaixo a lógica da função *add.Aresta()*, lembrando que o grafo é biderecional :

**Result**: Adiciona a aresta bidemensional no grafo com a modelagem da função custo
**Data**: Grafo G ;int Vertice1; int Vertice2; float probIncendio
float cost = (-1)*log(1.0 - probIncendio);
add.node(G.listaAdjacencia[Vertice1], Vertice2, cost);
add.node(G.listaAdjacencia[Vertice2], Vertice1, cost);

**Algorithm 2:** Pseudo-código add.Aresta()

#### 2.2.3 marcaPermitido()

Essa função é uma **DFS** que caminha ate a distância máxima permitida e marca todos os vértices com uma *flag* que permite o Dijkstra saber quais vértices são válidos e quais não.

**Result**: Marca os vertices do grafo que estão dentro da distância máxima do corpo de bombeiros
**Data**: Grafo G; int verticeBombeiros; int maxDistBombeiros;
Fila F;
F.enfilera(verticeBombeiros);
int dist=0;

**Algorithm 3:** Pseudo-código add.Aresta()

# 3 Análise de Complexidade Temporal e Espacial

Para simplificar, não serão demonstradas todas as análises de complexidade, mas demonstraremos algumas e mostraremos todas na tabela que segue abaixo:

| | NomeFuncao | ComplexidadeDeTempo | ComplexidadeDeEspaco |
|---|---|---|---|
| 1 | marcaDentroRaio() | O( \|V\|^2) | O( \|V\|^2) |
| 2 | addToPriorityQueue() | O(1) | O(1) |
| 3 | popMinCostSum() | O(\|V\|) | O(1) |
| 4 | Dijkstra() | O(\|V\|^2) | O(\|E\|log\|E\|+\|V\|) |
| 5 | initGrafo() | O(\|V\|) | O(\|E\| + \|V\|) |
| 6 | liberaGrafo() | O(\|V\|) | O(1) |
| 7 | printCaminho() | O(\|V\|) | O(\|V\|) |
| 8 | addAresta() | O(1) | O(1) |
| 9 | addNode() | O(1) | O(1) |
| 10 | liberaLista() | O(\|n\|) | O(1) |
| 11 | initFila() | O(1) | O(1) |
| 12 | isEmpty() | O(1) | O(1) |
| 13 | Empilha() | O(1) | O(1) |
| 14 | Queue() | O(1) | O(1) |
| 15 | deQueue() | O(1) | O(1) |
| 16 | liberaFila() | O(\|n\|) | O(1) |

## 3.1 Demonstração Complexidades

Iremos demonstrar apenas a complexidade das 2 principais funções:

- marcaDentroRaio()
- Dijkstra()

### 3.1.1 Complexidade marcaDentroRaio()

Para analisar a complexidade dessa função devemos começar percebendo que existem 3 loops dentro dela. Alem dos loops dentro da própria função temos ainda 2 funções que são chamadas por essa, Queue() e deQueue().

Iniciaremos analizando o primeiro loop que é um for, que quando analisamos a condição de parada têm complexidade $O(|V|)$,onde V é o número de vértices, tanto no melhor quanto no pior caso porque percorre todos os vértices. Os próximos 2 loops são compostos de 2 while (um dentro do outro) mais a chamada das 2 funções mencionadas.

É nítido que o pior caso para as funções deQueue() e Queue() é $O(|1|)$ para ambos os casos.

Por Ultimo analisamos os 2 loops while, onde é fácil imaginal o pior caso do loop exterior que é quando todos os vértices são alcançáveis fazendo assim a complexidade ser $O(|V|)$ e no melhor caso que é quando ninguém está conectado ao corpo de bombeiros que é $O(|1|)$. Analisando para o loop interior temos o pior caso ocorre quando um vértice está conectado a todos os outros vértices, fazendo assim esse loop ter complexidade $O(|V|)$.

Considerando tando agora a equação global das complexidades obtemos $O(|V|) + O(|V|) * (O(|V|) * O(|1|) + O(|1|)) = O(|V|^2)$.

### 3.1.2 Complexidade Dijkstra()

Quando analisamos o algorítimo de Dijkstra(), percebemos que ele é exatamente igual a busca em largura, a diferença é que ele só caminha pelos vertices permitidos e que seu Queue têm prioridade.

No pior caso, que é quando todos os vértices são permitidos, a única diferença é a complexidade do Queue de prioridade, que é obviamente $O(|V|)$, mas o mesmo vértice só pode entrar uma única vez no Queue, assim dentro do algorítimo a complexidade de todas as repetições é de $log(V) * O(|V|) = O(|log(V) * V^2|)$, isso pois V(exterior) decresce a cada iteração.

Assim para montar a equação faremos uma configuração um pouco diferente, iremos separar a complexidade de um while dentro do outro da complexidade da chamada da função popMinCostSum() dentro do while.

Assim temos equação que segue : $O(|V|)+O(|V|)*(O(|V|))+log(V)*O(|V|) = O(|V|^2)$ .

## 3.2 Complexidade Rotina main()

Para descobrir a complexidade da rotina precisamos analisar as rotinas principais, assim como já fizemos, e também o código como um todo.

O pior caso ocorre quando todos os vértices são conectados uns aos outros e existe corpo de bombeiros em todos os vértices. Nesse caso a complexidade é definida pela equação $O(|V|) + V * O(|V|^2) + O(|V|^2) = O(|V|^3)$.

### 3.3 Análise Teórica do Espaço

Para fazer a análise do custo de espaço, focamos na principal estrutura de dados que é o grafo. Para esse problema modelamos o grafo como uma estrutura de 4 vetores de tamanho $O(|V|)$, onde um desses vetores é um vetor de listas onde cada célula representa uma aresta do vértice. Quando todos os vértice estão conectados a todos os outros vértices seu tamanho é $O(|V|^2)$, onde que cada um desses elementos representa uma aresta, ou seja $O(|V|^2) * (2 * float + 1 * int)$.

Ou seja no pior caso a complexidade espacial segue a equação : $O(|V|^2) * (2 * float + 1 * int) + O(|V|) * 2 * int + O(|V|) * 1 * float$ .

## 4 Análise Experimental

Para criar os casos experimentais, assim como é definido pela especificação como parte do trabalho, criei um pequeno script em liguagem python para gerar de forma rápidad e fácil os mesmo. O código seguem Abaixo:

```
import numpy as np
import random
import sys
import os

def CriaArquivo(numVertices, numArestas, numBombeiros,distMaximaBombeiro):

    NomeArq="Caso_" + str(numVertices) +"Vertices_" + str(numArestas)+"Arestas_"
    numCasos= 10
    verticeIn= 1
    verticeOut= numVertices-2

    File = open(NomeArq, 'w')

    StringNumCasos = str(numCasos) + "\n"
    File.write( str(StringNumCasos) )

    HeaderString = ""
    ArgsVec = [numCasos, numVertices, numArestas, verticeIn, verticeOut, distMaxi
    for arg in ArgsVec :
        HeaderString = HeaderString + str(arg) + "␣"
    HeaderString = HeaderString + "\n"

    for nCasos in range(0,numCasos):

        File.write(HeaderString)

        ProbsArray = np.random.uniform(0,1,numArestas)
        SeqVertices = range(0,numVertices)
```

```
        for nArestas in range(0,numArestas):

            val1 = random.choice(SeqVertices)
            val2 = random.choice(SeqVertices)
            while(val1 == val2):
                val2 = random.choice(SeqVertices)

            StringAresta = str(val1)+ " " + str(val2) + " " + str( round( random
            File.write(StringAresta)

        Bombeiros = random.sample( range(0,numVertices), numBombeiros)
        for Bombeiro in Bombeiros:
            StringBombeiro = str(Bombeiro) + "\n"
            File.write( StringBombeiro )

    File.close()


def main():
    args = sys.argv[1:]
    if len(args) != 5:
        print "Erro tamanho Invalido"
        print str(args[0])
        sys.exit(1)

    CriaArquivo(args[0], args[1], args[2], args[3])
    print("Arquivo criado com sucesso")

if __name__ == '__main__':
    main()
```

Os casos teste que descidi criar foram seguindo 4 testes gerais:

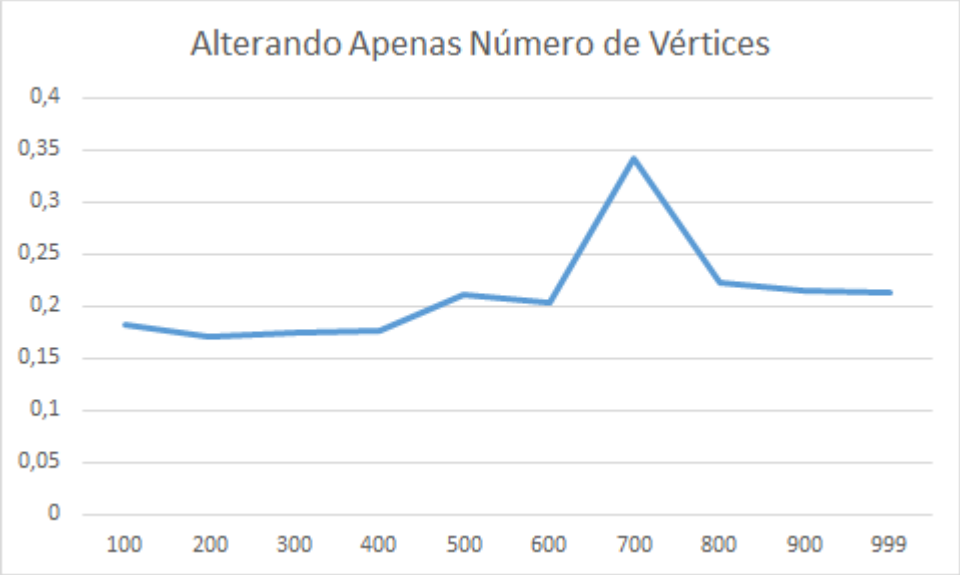
- Caso altera número de Vértices;
- Caso altera número de Arestas;
- Caso altera número de Quarteirões com Corpo de Bombeiros;
- Casos Extremos.

Para todos esses testes criei pelo menos 6 sets de parametros diferentes onde cada um desses sets contêm 10 instâncias, para análisar o tempo compilei a média de testes do mesmo arquivo.

## 4.1 Caso altera número de Vértices

Nesse teste fizemos uma configuração onde todos os grafos têm 10 mil arestas 10 quarteirões com corpo de bombeiros e podem percorrer distâncias até 10 quarteirões dos mesmos.

Então variamos apenas a quantidade de vértices do grafo, mas sem alterar nenhum dos outros parâmetros. O resultado inicial desse teste me assustou pois pensei que ele não fazia sentido.




Apos calmamente analisar o teste percebi que como todas as arestas são geradas randomicamente o número de vértices não diz respeito ao caminho e tão pouco a distância entre os vértices de entrada e saída.

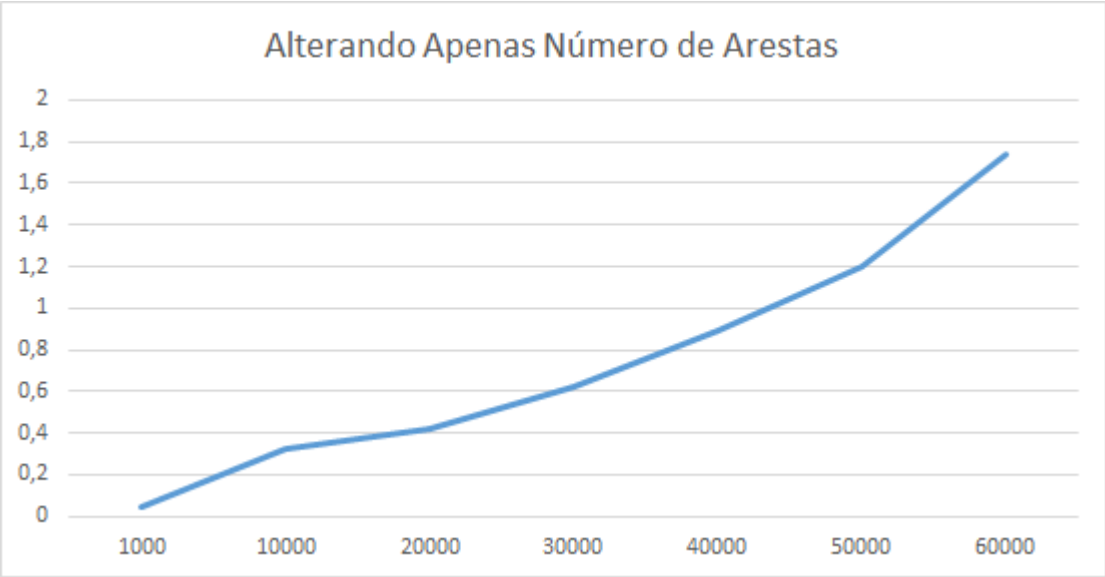
Assim se o grafo é esparso, o que é verdade para os grafos com maior número de arestas, sua complexidade não diz respeito a quantidade de vértices.

O motivo disso é importante ser ressaltado, no caso isso se deu por conta de uma decisão de projeto em implementar uma lista de adjacência ao invés de uma matriz de adjacência.

## 4.2 Caso altera número de Arestas

Esse caso é praticamente o inverso do caso anterior, no caso anterior analisamos o comportamento da complexidade de tempo em função do quão esparso é o grafo, nesse teste temos a análise da complexidade de tempo a medida que o grafo se torna mais completo.

Isso foi feito de forma onde todos os parâmetros são constantes e apenas o número de amostras que aumenta, como o gráfico abaixo mostra:

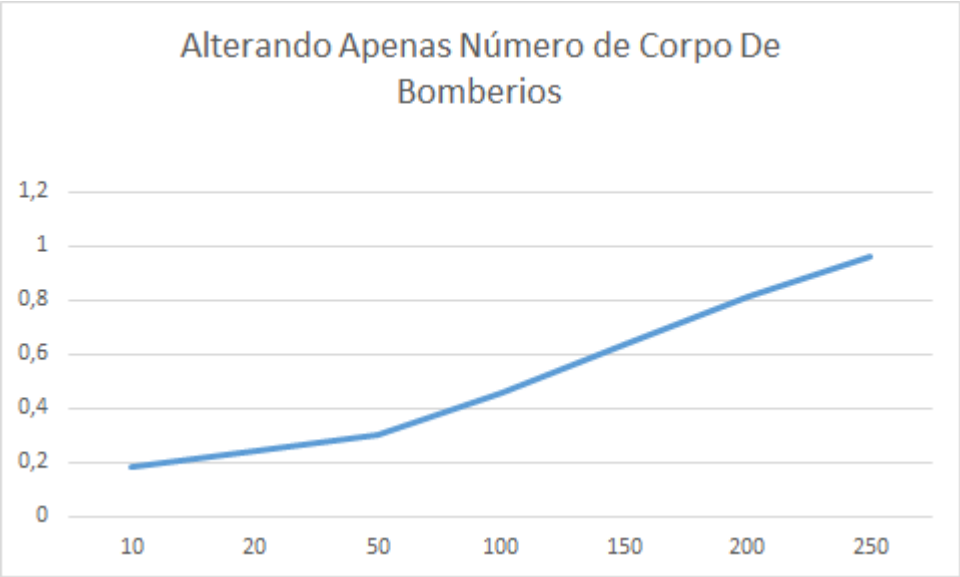



Percebemos pela imagem uma pequena formação de um gráfico de função quadrática, onde isso é devido algorítimo de Dijkstra.

De forma geral isso mostra que o nosso algorítimo é completamente influenciado por quão esparso o grafo é e isso é novamente por conta da forma como implementamos com listas de adjacência .

### 4.3 Caso altera número de Quarteirões com Corpo de Bombeiros

Esse gráfico mostra que a relação de aumentar apenas a quantidade de postos de corpo de bombeiro têm relação linear, e apenas linear.

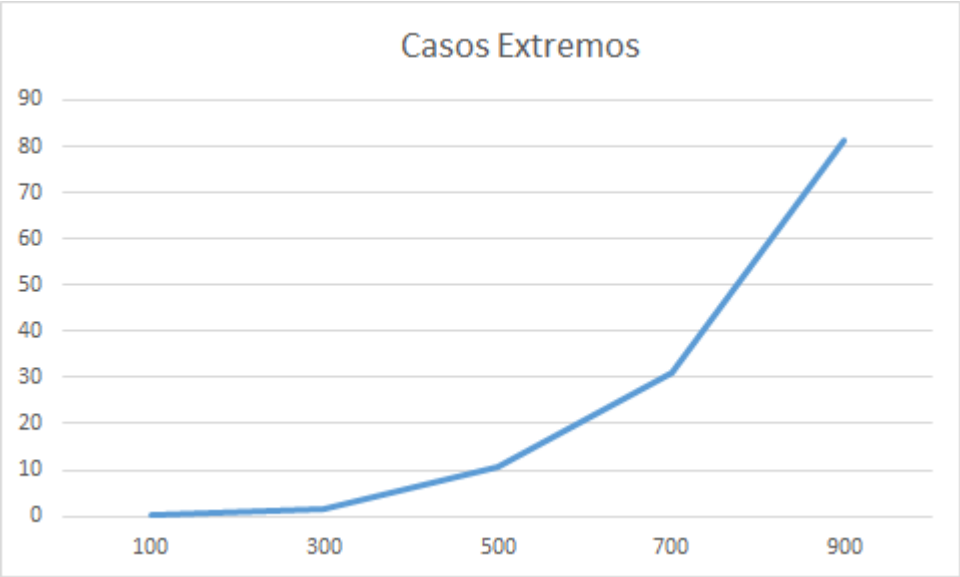



Mas o mesmo é verdade para quando aumentamos a distância máxima permitida de cada posto, assim quando combinamos as duas no pior caso que a complexidade se torna quadrática.

## 4.4 Caso Extremo

Esse é possivelmente o teste mais importante na análise de complexidade do nosso algorítimo, isso se deve ao fato onde cada um dos arquivos de entrada para esse teste representam 10 instâncias onde todas são o pior caso para o seu número de arestas.

Podemos perceber nele a complexidade prevista pela análise teórica do programa:




Nesse caso o gráfico teve uma inclinação quadrática muito grande porque tanto o número de quarteirões com bombeiros quanto os números de vertices e arestas estão crescendo, além é claro da distância máxima permitida dos postos de bombeiro.

Esse gráfico foi feito para as piores situações para comprovar a robustez do algorítimo e a nossa predição de complexidade.

# 5 Conclusão

Neste trabalho, foi resolvido o problema do bombeiro preguiçoso.O problema foi resolvido modelando-o como grafos e aplicando algoritmos Dijkstra e *Deapth Search First*(DFS) que foram aprendido em sala de aula. Isso mostrou a importância e versatilidade da teoria dos grafos e o quão poderosa essa ferramenta pode ser.

Alem disso percebemos que o problema foi resolvido e o mesmo teve exatamente a complexidade prevista para os casos extremos, o que mostra novamente que um algorítimo funciona exatamente da forma como ele foi projetado e não terá resultados melhores do que aqueles para os quais ele foi pensado.

# 6 Ressalvas

Gostaria de pedir decupas tanto pelo meu código em python como pela formatação desse trabalho, no momento estou tentado aprender LATEX e python assim achei que a melhor forma de aprender essas duas poderosas ferramentas seria na prática.