

Documentação do Trabalho Prático ZikaZeroZ

Alexandra da Silva Pereira¹

¹ Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brazil

alesilva241@gmail.com

1. Apresente uma modelagem para o problema ZikaZeroZ empregando grafos. Deixe claro quais as restrições e/ou suposições devem ser feitas.

No trabalho **ZikaZeroZ** partimos da suposição de que todos os grafos de entrada serão conexos e terão em seu conjunto de vértices todos os focos. Com isso teremos possíveis soluções com uniões de vértices , que estão no conjunto principal, e formam um subconjunto de vértices, que compõe um novo grafo conexo que possui cobertura para todo o conjunto de vértices.

No modelo existe uma classe grafo, onde se encontram todas as operações efetuadas sobre o grafo. Existem também uma outra classe chamada voluntário e o script main, em voluntário define-se o seu conjunto de laços de amizade e seu conjunto de focos conhecidos, já no script main compõe o corpo principal do programa, aciona a montagem do grafo, recepciona a solução e faz a sua escrita no arquivo de saída. Cada vértice do grafo é do tipo voluntário.

Uma informação importante é que em *python* os *arrays* são indexados a partir de 0, logo, todo meu conjunto de dados segue de $0, 1, \dots, n - 1$ e no fim da execução passa por um ajuste, no qual os vértices solução são adequados com $+1$. Por último, o programa assume que os grafos são não-direcionados, portanto as entradas (A, B) e (B, A) são consideradas apenas uma aresta e ambas adicionadas a lista de adjacência.

2. Descreva uma algoritmo para resolver o problema ZikaZeroZ, tendo em vista a modelagem realizada no Exercício 1.

Após algumas análises e discussões determinou-se que seria usada uma estratégia de *brute force* para solução do problema, pois a solução necessitava ser ótima. Como pré-requisito, neste caso, é necessário percorrer todo o espaço de soluções possíveis, ou seja, verificar tudo que é possível resposta.

Já partindo do suposto que é um problema trabalhoso e que custa caro, utiliza-se lista de adjacências buscando uma economia em complexidade espacial, principalmente porque os grafos de entrada não são muito densos, em sua maioria. Entretanto, caso sejam densos a complexidade espacial é equivalente à matriz.

Inicialmente é feita a montagem do grafo, com o modelo descrito no item anterior. Na classe *graph.py* foram desenvolvidos métodos para operar sobre o grafo em si. Todos os métodos que operam sobre o grafo estão nesta classe, ou seja, operações de adicionar vértices, arestas, fazer cópia do grafo, uma busca em profundidade para verificar conexidade e finalmente, um método para buscar o melhor subconjunto de vértices que cobre todos os focos, e que será a resposta do problema.

Na classe *vonluntary.py* atribuimos tudo que pertence a cada voluntário, ou seja, a sua lista de laços de amizade, e uma lista de focos conhecidos por aquele voluntário. Além disso, para facilitar análises foi necessário incluir um campo nodeId a este modelo, para que fosse utilizado na cópia dos grafos que tem possíveis soluções. A classe *main.py* é responsável pela leitura do arquivo e criação dos objetos do tipo grafo e do tipo voluntário. Além disso é feita uma chamada do método *coberturaFocos*, que retornará a resposta para ser escrita em um arquivo.

A seguir são apresentados o pseudo-códigos de alguns métodos.

input : número de vértices, número de focos
output : subconjunto solução

```

1 instanciação de variáveis;
2 for i  $\leftarrow 2^n$  to 1 do
3     grafoAuxiliar  $\leftarrow$  copiaMáscara();
4     foreach grafoAuxiliar do
5         if se já houver possível solução then
6             if verifiqueQtdVertices() < Solução then
7                 if verifiqueSeCobreTodosFocos() then
8                     delete vértices não usados na cópia;
9                     buscaProfundidade();
10                    if verifiqueConexidade() then
11                        copieSolução;
12                    end
13                end
14            end
15        else
16            copieSolução;
17        end
18        limpeGrafoAuxiliar();
19    end
20 end
21 return Solução;
```

Algoritmo 1: Método *coberturaFocos()*

No bloco referente ao *algoritmo 1* apresentamos a sequência de operações necessárias para descobrirmos qual o subconjunto de vértices será a solução. A ideia geral é gerar um binário que é utilizado para formação dos subconjuntos de 1 a 2^n . Da seguinte forma:

- 11111 - Conjunto de vértices: 1, 2, 3, 4, 5
- 11110 - Conjunto de vértices: 1, 2, 3, 4
- 11101 - Conjunto de vértices: 1, 2, 3, 5
- ...
- 00010 - Conjunto de vértices: 4
- 00001 - Conjunto de vértices: 5

A partir disso verificamos se esse subconjunto é melhor que a solução atual, caso seja, verificamos se o conjunto cobre todos os focos, se ele é conexo com uma busca em profundidade. No fim, caso todas restrições sejam satisfeitas é feita uma atualização do grafo solução.

```

input : valor entre  $2^n$  e 1
output: copiaGrafo

1 crie um grafo para copia;
2 gere o binário para N;
3 for  $i \leftarrow 1$  to númeroVértices do
4   | copie o vértice i e suas arestas para a cópia, caso o bit i de N seja 1;
5 end
6 return copia;

```

Algoritmo 2: Método copiaMascara()

Já o *algoritmo 2* descreve a estratégia de receber o binário e criar uma cópia do grafo original baseando-se nos bits 1 de cada uma das combinações existentes de 1 a 2^n .

3. Analise as complexidades Temporais e Espaciais (usando notação assintótica) do algoritmo proposto.

Nesse problema, é razoavelmente fácil notar que a complexidade temporal é da ordem de 2^n , porque o loop principal deve percorrer todos os possíveis subconjuntos de vértices do grafo de entrada; por teoria de conjuntos, o número de subconjuntos de V é 2^V , logo isso implica em uma complexidade de $O(2^n)$ no pior caso. O pior caso é quando a única solução possível é o próprio grafo de entrada, logo o algoritmo deve percorrer todos os subconjuntos presentes. O melhor caso é quando todo conjunto de focos é coberto por apenas um vértice, isso porque, não serão executadas todas as operações como por exemplo: *busca em profundidade, verificação de conexidade, troca da solução* de maneira exaustiva. Essas operações só serão executadas uma vez em apenas um grafo de tamanho $n, n - 1, n - 2, n - 3\dots$ até o subgrafo ótimo de tamanho 1, o que torna o complexidade $O(n)$. Já no caso da complexidade espacial, o programa trabalha sobre uma lista de adjacência, logo isso representa em memória a quantidade de *bits* necessária para armazenar $(V + E)$, que é $(N + M)$ em nosso contexto, e representam o número de vértices e arestas respectivamente. No entanto possuímos uma instância inicial e uma cópia, no qual são feitas todas as operações, logo a notação assintótica que representa a complexidade espacial é $O(2(N + M))$.

4. Implemente o algoritmo proposto no Exercício 2 na linguagem de programação C, C++, Java ou Python.

A implementação segue no seguinte conjunto de arquivos: *graph.py*, *voluntary.py* e *main.py*. Além disso foram inclusos os arquivos de entrada e saída, respectivamente denominados *in* e *out*, bem como, novos grafos gerados pelos scripts *randomgraph_clustervertex.py* e *randomgraph_singlefocus.py*. Os arquivos geram respectivamente grafos com todos os focos no último vértice e grafos onde cada vértice cobre um foco.

5. Execute testes da implementação do Exercício , propondo instâncias interessantes para o problema ZikaZeroZ. Uma sugestão é que seja produzido um gráfico de Tempo de execução por Tamanho da entrada(n, m, r). Outra sugestão é relatar o tamanho da maior instância para o qual foi produzida uma solução.

Foram executadas várias instâncias produzidas com um script python objetivando criar grafos com maior número de vértices para que fosse possível observar o comportamento exponencial do algoritmo desenvolvido. Nas figuras 1 e 2 é possível observar a execução do script python main.py com a passagem dos parâmetros de entrada e saída do tempo de execução de cada instância. Na figura 3 apresentamos o gráfico de Tempo de Execução pelo Número de Vértices. No gráfico temos a representação de (n, m, r) para cada um dos pontos plotados. Nota-se que o valor que mais influencia o tempo de execução é o número de vértices por $O(2^n)$.

```
x alexandra@alexandra:~/Dropbox/UFMG/MASTER-DEGREE/1TH SEM/PAA/UFMG/TPI> for n in 0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 18 19 20
do
python main.py in$n out$n
done
(0.012048006057739258, 'seconds')
(0.023566007614135742, 'seconds')
(0.036428213119566836, 'seconds')
(0.0779118537902832, 'seconds')
(0.004729032516479492, 'seconds')
(0.11727404594421387, 'seconds')
(0.009192228317260742, 'seconds')
(0.010195016860961914, 'seconds')
(0.381072998046875, 'seconds')
(0.7362370491027832, 'seconds')
(1.5134799480438232, 'seconds')
(3.27856707572937, 'seconds')
(6.853554964065552, 'seconds')
(16.007235050201416, 'seconds')
(31.867379903793335, 'seconds')
(68.07764911651611, 'seconds')
(150.7258858680725, 'seconds')
(293.44757890701294, 'seconds')
(696.304291009903, 'seconds')
```

Figura 1. Saídas que representam o tempo de execução das novas instâncias.

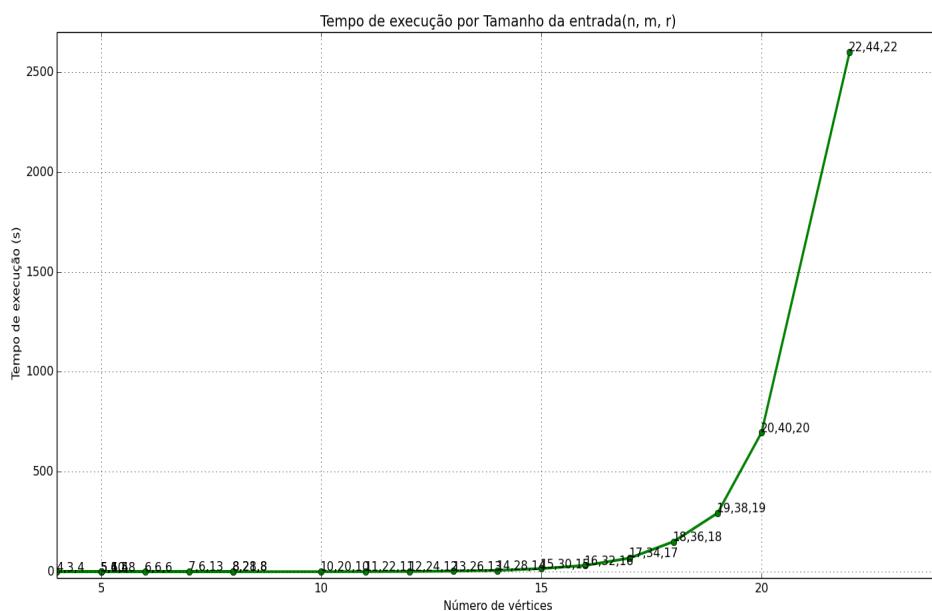


Figura 2. Tempo de Execução x Número de Vértices

Em tentativas de executar grafos com mais de 23 vértices aguardou-se a execução do algoritmo por cerca de 50 minutos, sem sucesso. O tempo estimado é de 1h30min.

6. Compare a análise e a execução, respectivamente, Exercícios 3 e 5.

Principalmente, relate se as previsões teóricas estão em concordância com os resultado experimentais.

Como esperado, as previsões teóricas foram confirmadas com os resultados experimentais. Observando as figuras podemos ver que o tempo de execução para n vértices é aproximadamente igual ao dobro do tempo de execução para $n - 1$, logo $t(n + 1) = 2 * t(n)$, e se o tempo de n é $O(2^n)$ como previsto no exercício 3, logo $t(n + 1) = 2 * 2^n = 2^{n+1}$, portanto a análise teórica é comprovada por indução e apresenta crescimento exponencial.

Trabalho Prático: ZikaZeroZ

Samuel Moreira Abreu Araújo¹

¹Universidade Federal Minas Gerais (UFMG)

Belo Horizonte – MG – Brazil

samuelcco@gmail.com

1. Exercício - Modelagem do problema

A abordagem aqui proposta para o problema ZikaZeroZ foi modelada com base em um grafo, que pode conter ciclos, sem pesos nas arestas e sem arestas paralelas. Tal representação em grafos torna a visualização do problema mais simples e de fácil entendimento.

O grafo consiste em uma estrutura $G(V, E)$, onde:

- Cada vértice $n \in V$, representa um conjunto de voluntários, que pode alcançar r focos.
- Cada foco $r \in F$, pode variar de $1, 2, 3, \dots, nr$ focos.
- Cada aresta $m \in E$, entre (u, v) , representa o laço de amizade entre os voluntários.

Devido à peculiaridade da solução aqui proposta, o grafo foi representado usando uma matriz de adjacências. Como não existem *self loop* a diagonal principal é sempre constituída de 0 e como é uma realação não dirigida das arestas, a matriz sempre será espelhada, como pode ser visto na figura 1.

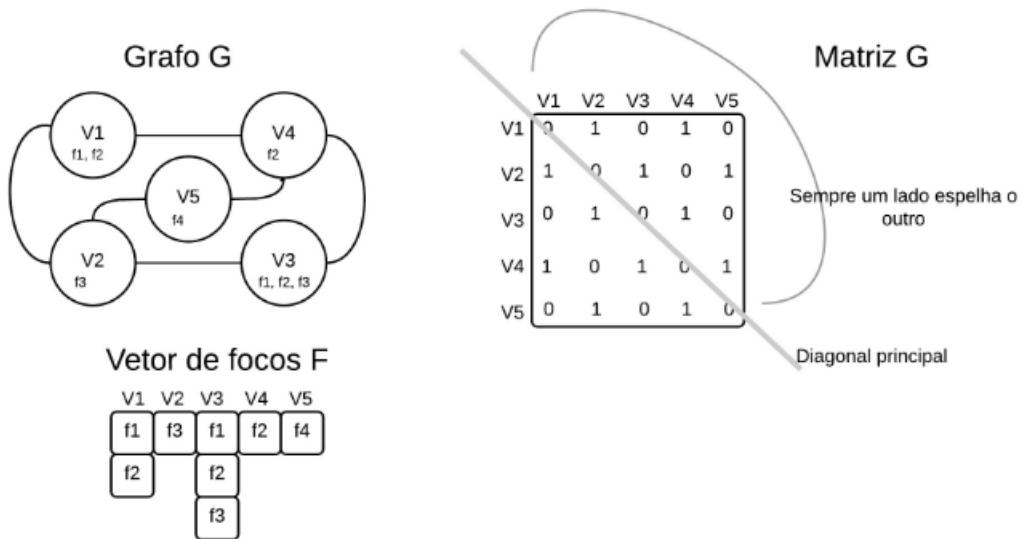


Figura 1. (a) Grafo,(b) Matriz de adjacência e (c) Vetor de focos.

Em nossa representação de dados, os vértices são numerados $1, 2, 3, \dots, |V|$, onde $|V|$ o numero total de nós. A matriz de adjacência consiste em uma estrutura $|V| \times |V| A = (a_{ij})$, (exemplo figura 1(a)), tal que:

$$|a_{ij}| = \begin{cases} 1, & \text{se } (i, j) \in E, \\ 0, & \text{se } (i, j) \notin E. \end{cases}$$

Justificando a escolha pela matriz de adjacências, segundo [Cormen et al. 2009], a matriz além de ser uma implementação mais simples e tão eficiente quanto a lista de adjacências na maioria dos aspectos, possui uma grande vantagem contra a lista de adjacência, quando o grafo é não ponderado. Em termos de espaço de armazenamento a matriz de adjacência usa somente um bit por entrada (*unsigned short int*), ao oposto da lista de adjacência que usa uma palavra de memória e mais uma coleção de apontadores.

O vetor de focos é composto por um vetor principal. Assumindo que cada vértice possui pelo menos um foco de cobertura, o vetor de focos vai ter sempre n posições e pode ser indexado pelo índice de alocação. A cada novo foco inserido, uma nova célula é expandida no índice do nó, como pode ser visto na figura 1(c).

O modelo de solução do problema, foi aqui implementado baseado em uma busca exaustiva, onde ele não cria a solução de maneira construtiva, e sim verifica em tempo polinomial se cada solução proposta é viável, no caso se cobre todos os focos e é conexa. Para cada solução a ser verificada é construído um subgrafo G'' a partir de G , contendo as arestas que incidem sobre os nós resultantes das posições das possíveis soluções (mais detalhes na tabela 1). Ao total serão construídos $2^n - 1$ subgrafos G'' distintos. De todas as $2^n - 1$ soluções candidatas, apenas 1 é a melhor solução (caso exista), e as outras devem ser descartadas.

O objetivo é encontrar uma rede conexa G'' , com menor número de voluntários, que cubra todos os focos, em caso de empates, a rede com menor somatório dos índices dos nós deve ser mostrada, em caso de não haver tal rede, o algoritmo retorna a inviabilidade.

2. Exercício - Descrição do algoritmo

A busca exaustiva é um algoritmo simples e de uso abrangente, mas com custo computacional alto, neste caso em particular, de ordem exponencial. Ele consiste em enumerar todas as possíveis soluções candidatas e verificar se cada uma satisfaz as restrições do problema. O algoritmo de busca exaustiva sempre analisa todas as soluções, por isso é garantido que ele sempre vai encontrar o ótimo (caso existir solução viável).

Na implementação aqui desenvolvida, o algoritmo é baseado em uma busca exaustiva com melhorias que ajudam a diminuir o tempo de execução. Todas as possíveis combinações de soluções são geradas e depois testadas uma a uma. Caso a solução corrente G'' seja viável e melhor que a já existente G^* , ela é salva em G^* . Caso uma solução corrente de G'' que esteja sendo processada se torne pior que a melhor já existente G^* , interrompe-se o processo e inicia-se outra iteração (caso haja). Ao final a melhor solução é exibida G^* . Em caso de empate, ou seja $|G''| = |G^*|$ e $\text{sum}(G'') = \text{sum}(G^*)$, o algoritmo opta por armazenar a última solução achada, exemplo:

- $G^* \rightarrow V^* = \{v2, v3, v4\}$
 - $|G^*| = 3$
 - $\text{sum}(G^*) = 9$
- $G'' \rightarrow V'' = \{v1, v3, v5\}$
 - $|G''| = 3$
 - $\text{sum}(G'') = 9$

Neste caso $G^* \leftarrow G''$

A geração de combinações é feita com base em um contador binário que começa sua execução de 1 até $2^n - 1$, onde n é o número de vértices total. Ex: para $n=3$, teremos 7 combinações diferentes, a combinação 0 não precisa ser feita pois não é uma solução válida (não contém nenhum nó). Um modelo de geração de combinações pode ser visto na figura 1.

Contador (2^n)	Posições dos nós
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

Tabela 1. Modelo de combinações entre 3 nós, gera 7 potenciais soluções.

O código do algoritmo1 é o método central da solução, e é explicado em mais detalhes a seguir.

Algoritmo 1 Busca Exaustiva (G)

```

1:  $G^* \leftarrow \infty$ 
2:  $n \leftarrow Vertices(G)$ 
3:  $flag \leftarrow 0$ 
4: for  $U \leftarrow$  cada permutação de 1 até  $2^n$  do
5:    $G'' \leftarrow G - U$                                 #  $G''$  contém as arestas excluídas da solução
6:   if  $tam(G'') \leq tam(G^*)$  then
7:     if  $VerificaFocos(G'') == TRUE$  then
8:       if  $conexo(G'') == TRUE$  then
9:         if  $tam(G'') == tam(G^*)$  and  $sum(G'') \leq sum(G^*)$  then
10:           $G^* \leftarrow G''$ 
11:           $flag \leftarrow 1$ 
12:        else if  $tam(G'') < tam(G^*)$  then
13:           $G^* \leftarrow G''$ 
14:           $flag \leftarrow 1$ 
15:        end if
16:      end if
17:    end if
18:  end if
19: end for
20: if  $flag == 1$  then
21:   imprime ( $G^*$ )
22:   return TRUE
23: end if
24: return FALSE

```

- Na linha 1 do algoritmo1, a melhor solução é iniciada com infinito, após cada

iteração, se existirem soluções melhores ela é atualizada (linha 9 e 12).

- Na linha 4 do algoritmo1, para cada solução gerada, inicia-se o teste de viabilidade.
- Na linha 5 do algoritmo1, é gerado um subgrafo G'' , contendo os vértices que fazem parte da solução a ser testada.
- Na linha 6 do algoritmo1, é testado se a solução corrente é maior que a melhor já encontrada, caso seja a solução é descartada, pois se procura a menor existente.
- Na linha 7 do algoritmo1, é verificado se a solução corrente cobre todos os focos, caso sim continua a execução, caso contrário a solução é inviável e descartada.
- Na linha 8 do algoritmo1, é verificado se a solução é conexa, ou seja se todos os vértices estão conectados em um único grafo G'' , caso não seja solução é inviável e descartada, esta etapa é feita com o algoritmo2.
- Na linha 9 do algoritmo1, é testado o caso de duas soluções serem viáveis e de mesmo tamanho, neste caso, opta-se por salvar a que contenha menor somatório dos índices dos nós utilizados.

Algoritmo 2 conexo (G'')

```

1: npreto  $\leftarrow 0$ 
2: for each  $v \in G''$  do
3:    $v.cor \leftarrow b$ 
4: end for
5:  $v \leftarrow$  qualquer verticealeatorio  $\in G''$ 
6:  $v.cor \leftarrow c$ 
7: insere ( $v, S$ )
8: while  $S \neq \emptyset$  do
9:    $u \leftarrow$  primeiroelemento( $S$ )
10:  for each  $v$  adjacente  $u \in G''$  do
11:     $v.cor \leftarrow c$ 
12:    insere ( $v, S$ )
13:  end for
14:   $u.cor \leftarrow p$ 
15:  np++
16: end while
17: if  $np == numVertices (G'')$  then
18:   return TRUE
19: end if
20: return FALSE

```

3. Exercício - Análise de complexidade de tempo e espaço

3.1. Análise de tempo

A análise de complexidade de tempo, não representa o tempo de relógio gasto em si [Cormen et al. 2009], mas sim o número de vezes que determinada tarefa é executada. Baseado nas estruturas de dados aqui utilizadas, a complexidade de tempo é dada pelos seguintes métodos:

- Transformação de inteiro para binário, método abstruído na linha 4 do algoritmo1. Preenche um vetor de n posições, sua complexidade é dada por ter que percorrer este vetor uma vez e fazendo divisões. Complexidade $O(n)$.

- Geração de soluções, linha 4 do algoritmo1. executa sempre $2^n - 1$ vezes. Sua complexidade é dada pelo número de saídas geradas, o que dá a característica exponencial do problema. Complexidade $O(2^n)$, onde para cada saída é executada uma vez o restante do código, esta função é que gera a complexidade final do algoritmo.
- Copia da matriz $m.res[][]$, linha 5 do algoritmo1. Realizada uma vez a cada iteração, faz a troca somente as linhas e colunas alteradas na última iteração. As trocas são feitas e uma matriz $n \times n$ o que gera a complexidade $O(n^2)$, mas no caso como a matriz é espelhada (ver figura 1(b)), as trocas podem ser feitas espelhando as posições, logo na metade do tempo. Complexidade $O(n^2)/2$.
- Verifica focos, linha 6 do algoritmo1, executada uma vez a cada iteração, faz a verificação para cada nó se ele acessa qual foco r . Complexidade no pior caso $O(n * r)$
- Tam (G''), linha 7 do algoritmo1, executada uma vez a cada iteração, não é necessário executar para G^* pois depois de uma vez executada, o valor fica salvo em uma variável tam . Complexidade $O(n)$
- Conexo (G''), chamada na linha 8 do algoritmo1, executa o algoritmo 2. Onde:
 - Percorre os n vértices e colore de branco. Complexidade $O(n)$
 - Muda cor vertice v , indexado na estrutura. Complexidade $O(1)$
 - Insere(v, S), insere no final. Complexidade $O(1)$
 - Retira(u, S), retira o primeiro. Complexidade $O(1)$
 - Verifica todos adjacentes $v \ adj u \in G$, como feito em matriz de adjacência, percorre-se toda a linha indexada do nó. Complexidade $O(n)$.
 A complexidade final da rotina é $O(n^2)$, pois para dizer se um nó é adjacente, deve se atravessar toda a matriz $m[][]$, que é feito no tempo $O(n)$.
- Sum (G''), linha 9 do algoritmo1, executada uma vez a cada iteração, não é necessário executar para G^* pois depois de uma vez executada, o valor fica salvo em uma variável tam . Complexidade $O(n)$

Ainda são feitas outras comparações de valores, mas nada que influencie na complexidade final do algoritmo. Assim pela análise de complexidade de tempo, pode-se dizer que o algoritmo executa em tempo exponencial da ordem de $O(2^n * (n^2 + nr))$, o mesmo pode ser verificado na figura 8.

3.2. Análise de espaço

A análise de complexidade de espaço mensura a quantidade de memória que determinados trechos de código utilizam para resolver determinada ação. É o gasto de memória utilizado para alocar determinada estrutura de memória. Em nosso algoritmo ela é destacada nos seguintes casos:

- Matriz principal $m.adj[][]$. Forma representativa usada para guardar os enlaces usados existentes entre os nós, alocada somente uma vez no programa, declarada com *unsigned short int*, e preservada com os elementos originais carregados do arquivo (declarada como G no pseudocódigo). Complexidade $O(n^2)$.
- Matriz secundária $m.res[][]$. Forma representativa usada para guardar os enlaces usados existentes entre os nós, alocada somente uma vez no programa, declarada com *unsigned short int*, sofre alterações de elementos a cada iteração, mas não de espaço (declarada como G'' no pseudocódigo). Complexidade $O(n^2)$.

- Vetor $vFoco[]$. Estrutura usada armazenar os focos existentes. Cada célula possui um *unsigned short int* que varia de acordo com o número máximo des focos, Complexidade $O(f)$.
- Vetor $f[][]$. Estrutura usada para indexar nós e acessar os focos. Os nós são acessados pelo índice do vetor, os focos são declarados como *unsigned short int*, cada nó possui um vetor de r focos (exemplo na figura 1(c)). Complexidade $O(sum(nf))$, onde $sum(nf)$ é o somatório total de todos os focos.
- Vetor $vert[]$. Estrutura usada na função de verificação se os nós são conexos. Cada célula possui um char para armazenar a cor e um *unsigned short int* para o índice. Complexidade $O(n)$.
- Vetor $S[]$. Estrutura usada na função de verificação se os nós são conexos. Os nós são armazenados em uma fila FIFO, cada célula possui um *unsigned short int* para o índice do nó colocado na fila (declarado como S no algoritmo2). Complexidade $O(n)$.
- Vetor $starPath[]$. Estrutura usada para armazenar a melhor solução encontrada. Cada célula possui um *unsigned short int* que varia entre 0 e 1 (declarado como G^* no algoritmo1). O vértice é acessado pelo índice de alocação. Complexidade $O(n)$.
- Vetor $n[]$. Estrutura usada para armazenar os valores gerados para cada solução a ser verificada (declarado como U no algoritmo1). A cada iteração o vetor é sobreescrito. Cada célula possui um *unsigned short int* que varia entre 0 e 1, o vértice é acessado pelo índice de alocação. Complexidade $O(n)$.

Ao final temos que o custo de espaço utilizado pelo algoritmo é de $O(2n^2 + f + sum(nf) + 4n)$.

4. Exercício - Algoritmo (implementação)

O algoritmo foi implementado em C++ de forma procedural. C++ foi escolhida por ser uma linguagem rápida, simples, e produzir códigos bem eficientes. O código foi modelado em um arquivo *main.cpp*, responsável pela chamada principal do programa, e outro *aux.cpp* com as rotinas que completam o projeto.

O código segue em anexo, para executá-lo bastar ter instalado o g++, e fornecer os arquivos de entrada e parâmetros como indicado a baixo:

- Compilar
\$./compilar.sh
- Executar
\$./executar.sh entrada.txt saida.txt

Exemplo de execução

```
$ ./executar.sh exemplos/exemplo1.txt saida.txt
```

5. Exercício - Execução e testes

Os testes foram divididos em duas seções para serem melhor explicados. Na primeira seção são feitos testes de exemplos que demonstram a corretude da solução. No segundo são feitos testes para análise de carga e tempo.

5.1. Teste de corretude

Vários testes de corretude foram feitos, aqui serão mostrados alguns casos particulares para se exemplificar o funcionamento. Os casos maiores ficam inviáveis de serem mostrados graficamente, mas foram testados e verificados.

Nos testes a seguir (figuras 2, 3, 4 e 5) o algoritmo foi modificado para exportar todas as soluções válidas, sem fazer podas.

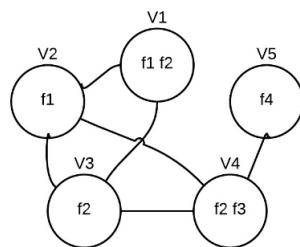


Figura 2. Grafo com 5 nós, 6 arestas e 4 focos diferentes.

Soluções viáveis encontradas pelo algoritmo 1, do grafo presente na figura 2:

1. : V2 - V4 - V5
2. : V1 - V2 - V4 - V5
3. : V1 - V3 - V4 - V5
4. : V2 - V3 - V4 - V5
5. : V1 - V2 - V3 - V4 - V5

Sendo que a melhor solução encontrada foi a 1, encontrada na iteração 26, tendo executado a rotina de verificação de conectividade 6 vezes.

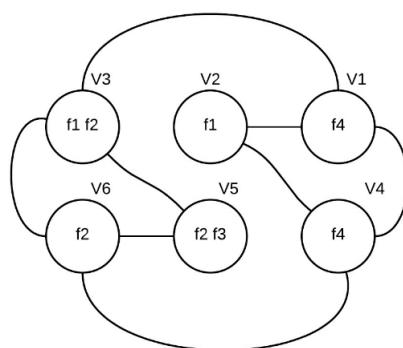


Figura 3. Grafo com 6 nós, 8 arestas e 4 focos diferentes.

Soluções viáveis encontradas pelo algoritmo 1, do grafo presente na figura 3:

1. : V1 - V3 - V5
2. : V1 - V2 - V3 - V5
3. : V1 - V3 - V4 - V5
4. : V1 - V2 - V3 - V4 - V5
5. : V1 - V3 - V5 - V6

6. : V1 - V2 - V3 - V5 -V6
7. :
12. : V1 - V2 - V3 - V4 - V5 -V6

Sendo que a melhor solução encontrada foi a 1, encontrada na iteração 21, tendo executado a rotina de verificação de conectividade 2 vezes.

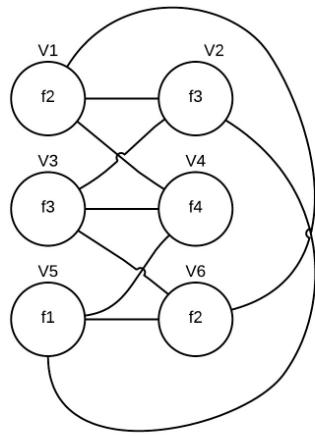


Figura 4. Grafo bipartite, com 6 nós, 9 arestas e 4 focos diferentes.

Soluções viáveis encontradas pelo algoritmo 1, do grafo presente na figura 4:

1. : V1 - V2 - V4 - V5
2. : V1 - V3 - V4 - V5
3. : V1 - V2 - V3 - V4 - V5
4. : V2 - V4 - V5 - V6
5. : V1 - V2 - V4 - V5 - V6
6. : V3 - V4 - V5 - V6
7. : V1 - V3 - V4 - V5 - V6
8. : V2 - V3 - V4 - V5 - V6
9. : V1 - V2 - V3 - V4 - V5 - V6

Sendo que a melhor solução encontrada foi a 1, encontrada na iteração 27, tendo executado a rotina de verificação de conectividade 1 vezes.

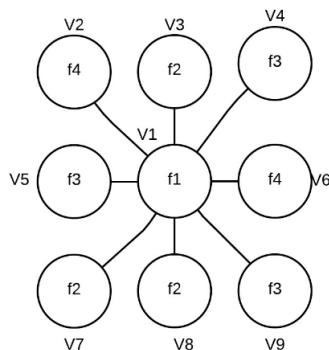


Figura 5. Grafo formato estrela, com 9 nós, 8 arestas e 4 focos diferentes.

Soluções viáveis encontradas pelo algoritmo 1, do grafo presente na figura 5:

1. : V1 - V2 - V3 - V4
 2. : V1 - V2 - V3 - V5
 3. : V1 - V2 - V3 - V4 - V5
 4. : V1 - V3 - V4 - V6
 5. : V1 - V2 - V3 - V4 - V6
 6. : V1 - V3 - V5 - V6
 7. : V1 - V2 - V3 - V5 - V6
 8. : V1 - V3 - V4 - V5 - V6
 9. : V1 - V2 - V3 - V4 - V5 - V6
 10.
511. : V1 - V2 - V3 - V4 - V5 - V6 - V7 - V8 - V9

Sendo que a melhor solução encontrada foi a 1, encontrada na iteração 15, tendo executado a rotina de verificação de conectividade 1 vezes.

Percebe-se que as melhores soluções G^* encontradas sempre são as primeiras a serem achadas. Mais testes na próxima seção irão mostrar que o contador começando a formar grupos de soluções de ordem crescente, acha-se a melhor solução ao começo das execuções, e dificilmente consegue-se melhorá-las, pois a tendência é que se use mais nós nas soluções seqüentes.

Usando a verificação de conectividade, a partir do menor vi presente na lista de vértices da solução corrente, acaba-se gerando a descoberta de outros vértices em ordem crescente, pois eles são lidos em seqüência de $1 \Rightarrow n$, do arquivo de origem. Logo não é preciso ordenar a solução, como pode ser visto nas saídas dos testes presentes das figuras 2, 3, 4 e 5 e nos testes de carga feitos.

5.2. Teste de desempenho

Nesta seção são apresentados os resultados computacionais obtidos através da abordagem algoritmica proposta com variações de cenários. Os testes foram realizados em um computador Intel Core i5 2^a geração, com 8GB de RAM DDR3, utilizando o sistema operacional Ubuntu 14.04.2 LTS. Os algoritmos foram testados através de 1 execução, por tratar-se de uma abordagem determinística.

As instâncias utilizadas na figura 6 e 7, foram geradas com um gerador próprio, usando uma distribuição uniforme de enlaces e focos entre os parâmetros testados. O numero de vértices n variou de 6 a 30, e os enlaces inseridos aleatoriamente entre $|V|$ e $2|V|$. O número de focos seguiu uma distribuição uniforme, variando de acordo com os parâmetros. Segundo [Walpole et al. 2011], a distribuição uniforme é caracterizada por uma função de densidade simples e plana, assim a probabilidade é uniforme de determinado item ser escolhidos em um intervalo fechado $[A, B]$, o que garante um número finito de resultados com chances iguais de acontecer.

Na figura 6, foram testadas três variações de cenários:

- 1f, cenário viável, com apenas um foco coberto por cada grupo de voluntários (1 foco por nó)
- 5f, cenário viável, com 5 focos cobertos por cada grupo de voluntários (5 focos por nó)

- 10f, cenário viável, com 10 focos cobertos por cada grupo de voluntários (10 focos por nó)

Tal cenário é proposto para comparar o quanto o aumento da cobertura de focos pode fazer com que o algoritmo demore mais a executar (figura 6). Pode-se observar que com um número maior de focos por nó, o tempo de processamento da solução aumenta. Este fato se dá por existir um maior número de focos por nó, logo maior a chance da solução ser válida. Ou seja, maior a chance de a execução passar da linha 7 do algoritmo 1. Apesar do aumento do tempo de processamento, o mesmo só é percebido acima de 18 nós, neste ponto a característica exponencial da solução faz com que todas as variações de cenários aumentem consideravelmente seu tempo de execução, como mostrado na figura 6.

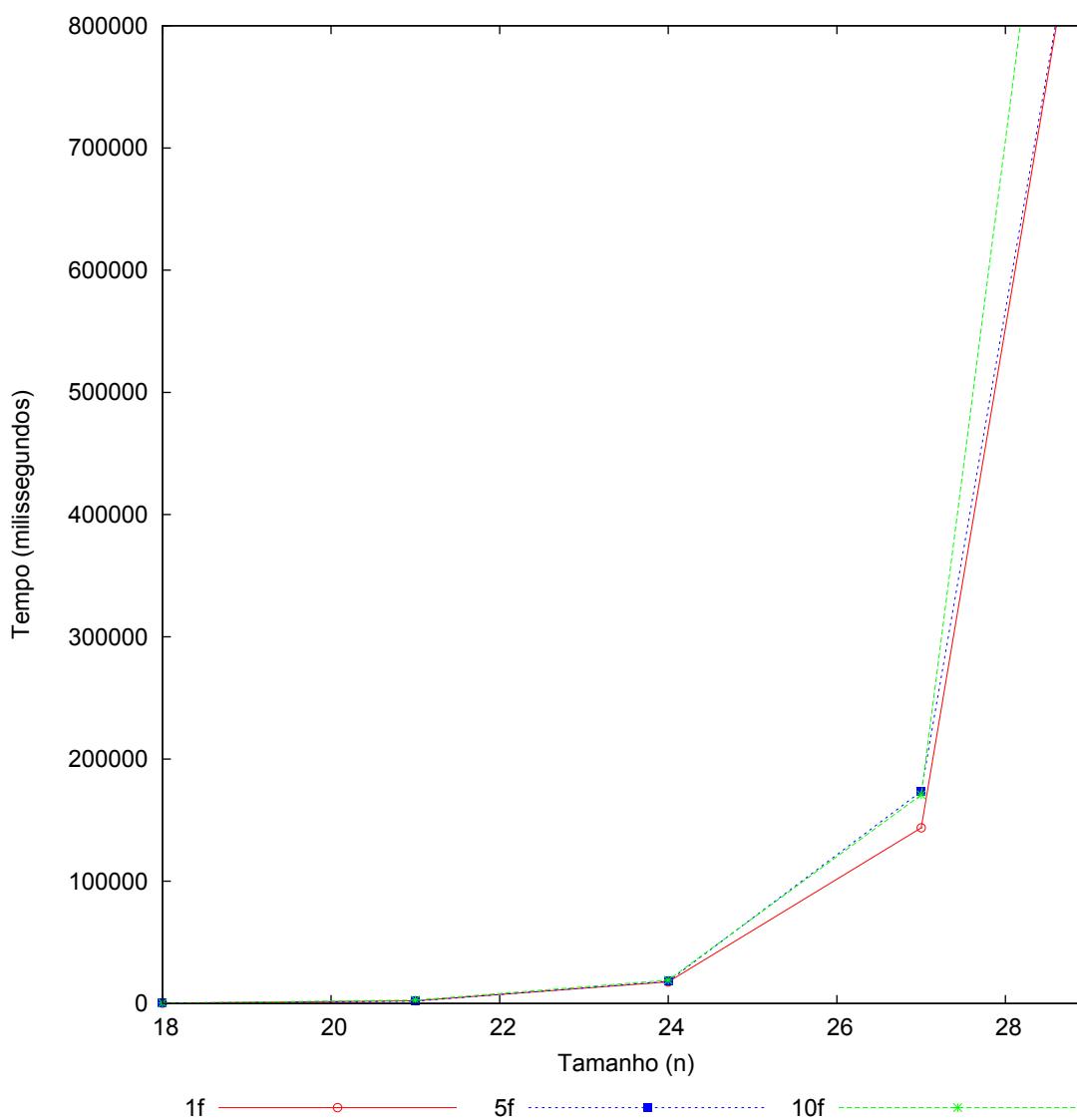


Figura 6. Testes com a variação do número de focos coberto por cada nó.

A variação de tempos mostrada na figura 6, demonstra que a poda, por verificação de cobertura de focos feita na etapa inicial, acaba melhorando o desempenho do algo-

ritmo, pois evita que varias soluções sejam processadas em um trecho mais custoso do código, já que são inviáveis.

Na figura 7 foram propostas alterações no modo de gerar as soluções, ou seja, alterações na geração de números do contador binário (linha 4 algoritmo 1). São propostas variações de se gerar as soluções de forma binária crescente, e de forma decrescente, começando com maiores sequencias "111..."e ir decrescendo, como pode ser observado nas tabelas 2 e 3 respectivamente.

solução (2^n)	Posições dos nós
1	0 0 1
2	0 1 0
...
2^n	1 1 1

Tabela 2. Exemplo de geração de soluções crescente para 3 nós.

solução (2^n)	Posições dos nós
2^n	1 1 1
2^{n-1}	1 1 0
...
1	0 0 1

Tabela 3. Exemplo de geração de soluções decrescente para 3 nós.

Ainda foram testadas três variações de tipo de cenário, onde:

- No primeiro caso a melhor solução é achada na primeira posição investigada, neste caso a maioria das outras soluções serão podadas na linha 6 do algoritmo1, pois temem a ser maiores.
- No segundo caso, nunca os voluntários vão cobrir todos os focos existentes, ou seja, não vai passar da linha 7 do algoritmo 1 (neste caso nunca vai existir solução viável).
- No terceiro caso, o garfo G é separado em dois sub grafos $G1$ e $G2$, onde cada um cobre metade dos focos, mas $G1$ e $G2$ são desconexos, logo nunca haverá uma solução válida. Neste caso a execução vai ser podada na hora de se verificar a conectividade, linha 8 do algoritmo 1 (o algoritmo de verificar conectividade vai descobrir apenas os vértices de um sub grupo).

Cada um dos três cenários descritos acima foi testado de forma crescente e decrescente, como mostrado na tabela 2 e 3. Explicando cada variação das nomenclaturas usadas na figura 7, segue:

- P_c = Primeira posição válida cobre todos os focos, com gerador soluções crescente.
- P_d = Primeira posição válida cobre todos os focos, com gerador soluções decrescente.
- S_c = Não cobre todos os focos, sem solução viável, com gerador soluções crescente.

- S_d = Não cobre todos os focos, sem solução viável, com gerador soluções decrescente.
- BSc = Grafo partido em dois sub grafos desconexos, solução inviável, com gerador soluções crescente.
- BSd = Grafo partido em dois sub grafos desconexos, solução inviável, com gerador soluções decrescente.

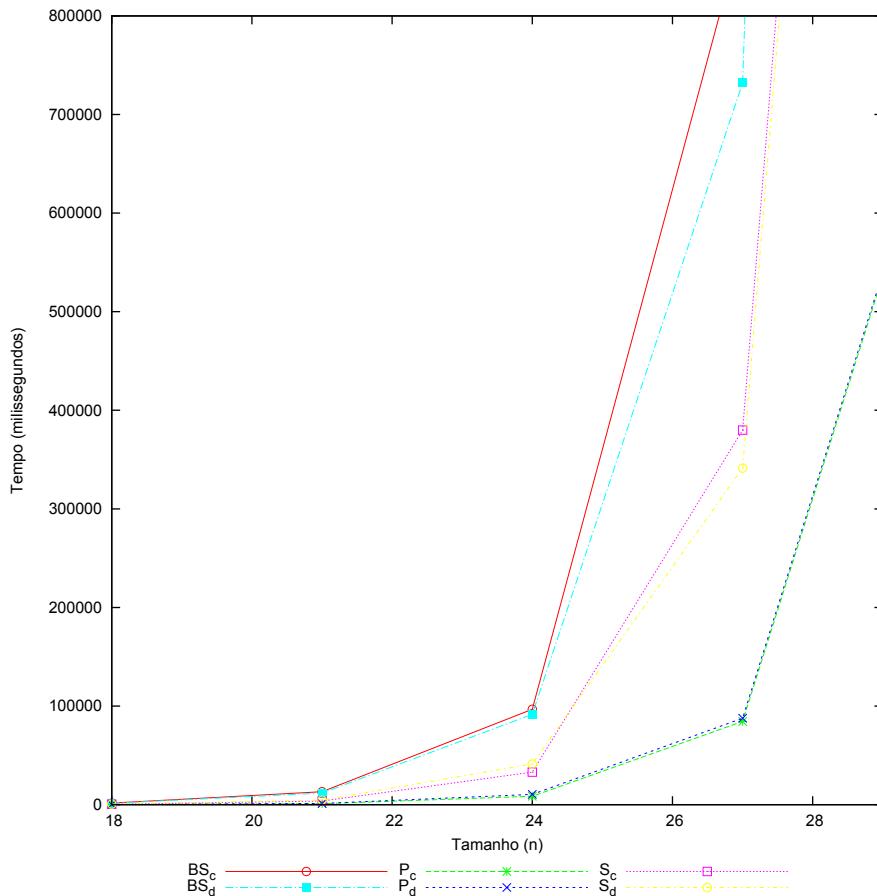


Figura 7. Testes com a variação da ordem da geração das soluções.

Os testes mostrados na figura 7 mostram que os casos de melhor execução para soluções válidas, se dão quando a solução é encontrada nas primeiras posições, e todas as outras soluções são podadas. Percebe-se que quando a solução é encontrada ao começo das iterações, tanto faz se o algoritmo funciona de maneira crescente ou decrescente, repare na figura 7 que os tempos de P_d e P_c sempre estão entrelaçados.

Ainda de acordo com a figura 7, o cenário abordado em BSd e BSc , além de não gerarem uma solução válida, tendem a ser mais lentos que os demais. Estes cenários se comportam assim, pois violam a maioria das podas (depende dos casos), e tem a execução interrompida somente no trecho de análise de conectividade. O BSd , consegue ser o pior de todos. Este fato indica que em cenários mistos, onde testa-se a conectividade na maioria das vezes, é melhor formar os grupos em ordem crescente de tamanho.

Os cenários para os casos sem solução válida por falta de cobertura de focos, tendem a ser mais rápidos que os que analisam a conectividade BSc e BSd , este fato

se dá por a análise de conectividade ser uma etapa mais lenta que a análise de cobertura de focos. Para estes casos a execução tem seu tempo de vida interrompido na verificação da linha 7 o algoritmo1, gerando a diferença.

6. Exercício - Comparação e análise da execução

Apesar do tempo de execução ser alto, o algoritmo mostrou-se bastante seguro e confiável nos testes feitos, além de bem comportado quanto ao consumo de memória. Tal método de busca exaustiva traz a certeza do resultado exibido ser ótimo pelo fato de estarmos vasculhando todas as possíveis soluções.

Para um número pequeno de vértices o algoritmo teve uma execução rápida, mas seu comportamento quando o número de vértices se torna relativamente maior, é claramente exponencial, como podemos ver no comparativo do gráfico 8, plotado com constantes $c1 = 1/500$ e $c2 = 1/1000$, onde os tempos do algoritmo proposto são limitados por $c(2^n)$.

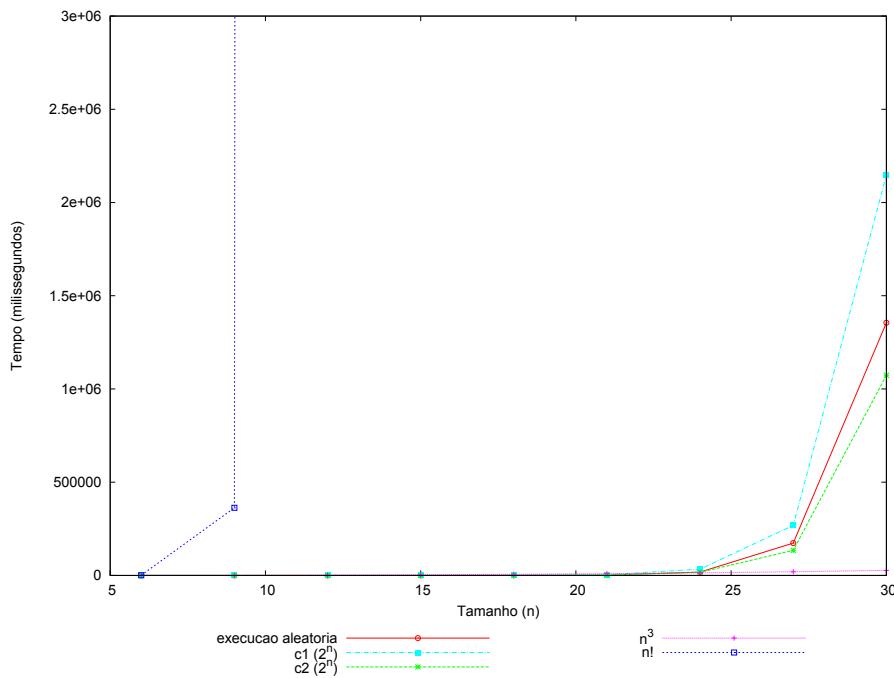


Figura 8. Comportamento assintótico de funções polinomiais e exponenciais, sendo $c1 = 1/500$ e $c2 = 1/1000$.

Este algoritmo é de baixa eficiência assintótica de tempo e inviável para ser executado em um cenário real, devido a ser de alto custo computacional, onde dependendo da instância de entrada, irá-se demorar muito para finalizar a execução. Mas acreditamos que as melhores soluções vão ser achadas nas primeiras iterações do código, onde as primeiras soluções achadas podem ser usadas como base para métodos de refinamento heurísticos, o que torna a abordagem interessante para o meio acadêmico.

Para trabalhos futuros acreditamos que heurísticas polinomiais possam conseguir bons resultados em tempos de execução viáveis.

Referências

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (2009). *Introduction To Algorithms*. MIT PRESS, 3th edition.

Walpole, R. E., Myers, R. H., Myers, S. L., and Ye, K. (2011). *Probability & Statistics for Engineers & Scientists*. Pearson, 9th edition.

PAA - Trabalho Prático 1: ZizaZeroZ

Thiago Carvalho D'Ávila¹

¹Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
Belo Horizonte – MG – Brazil

thiagoc@ufmg.br

1. Modelagem (Exercício 1)

Dado um conjunto universo \mathbf{U} de r focos do mosquito:

$$\mathbf{U} = \{f_1, f_2, f_3, f_4\}$$

(exemplo para r=4)

Dados também uma coleção \mathbf{S} de subconjuntos $s_i \subset \mathbf{U}$, dos focos cobertos por cada voluntário, e um conjunto de relações \mathbf{Q} entre os voluntários j e k (s_j, s_k) com $s_j, s_k \in \mathbf{S}$:

$$\mathbf{S} = \{\{f_1, f_2\}, \{f_1\}, \{f_2\}, \{f_2, f_3\}, \{f_4\}\}$$
$$\mathbf{Q} = \{(s_1, s_2), (s_1, s_3), (s_2, s_3), (s_2, s_4), (s_3, s_4), (s_4, s_5)\}$$

Pode ser modelado um problema de **Set-Cover** [Vazirani 2001] em que, dada uma função de custo $c : \mathbf{S}$, que é o número de subconjuntos de \mathbf{S} , deve-se encontrar a menor subcoleção de \mathbf{S} (ou menor custo) tal que a união dos subconjuntos cobertos possua todos os elementos de \mathbf{U} .

Além disso, existe a restrição de que, para uma subcoleção ser considerada uma solução válida, todo subconjunto s_i da subcoleção de \mathbf{S} deve ter uma relação em \mathbf{Q} com outro subconjunto da mesma.

Assim sendo, define-se um grafo $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ de forma que, $v \in \mathbf{V}$ é um vértice que modela um estado de cobertura de focos e possui uma subcoleção de subconjuntos $\in \mathbf{S}$. Já os arcos $e \in \mathbf{E}$ representam as transições de estado, dadas por uma função **sucessor**, que adiciona um novo subconjunto de \mathbf{S} respeitando as relações de \mathbf{Q} .

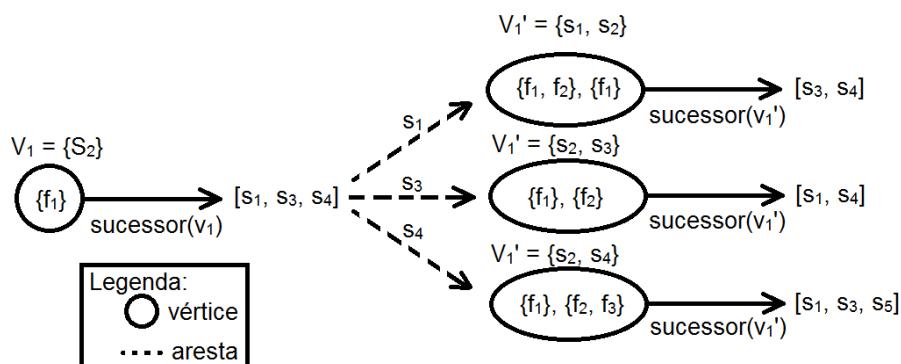


Figura 1. Aplicação da função sucessor a partir de um estado v_1 seguida de um estado v_1'

Considerando-se os conjuntos **U**, **S** e **Q**, exemplificados anteriormente, dado um vértice $v = \{s_2 = \{f_1\}\}$, a função **sucessor** (Fig. 1) deve retornar todos os possíveis subconjuntos $\in S$ que podem ser adicionados ao estado v e possuem relação em **Q** podendo originar um novo estado v' .

2. Algoritmo (Exercício 2)

Primeiramente, é lido o arquivo de entrada com **S**, **Q** e **r** (número de focos) gerando com esses dados, respectivamente, as matrizes **VOLUNTEER_COVER**, **RELATIONSHIP_MATRIX** e o **UNIVERSE_SET** (conjunto universo), que são variáveis globais para o algoritmo.

Em seguida, a solução do problema é inicializada como *Nenhuma* e o grafo é inicializado com o vértice contendo o subconjunto do primeiro dos voluntários. Isto ocorre, porque não se pode começar com a subcoleção vazia, já que, para um conjunto vazio, não há relação com nenhum outro subconjunto em **Q**, por consequência não existe sucessor para gerar os vértices do grafo.

Algoritmo 1: Breadth-First Search (BFS)

```

1 BFS(vertice_inicial)
2 Cria fronteira como uma fila vazia
3 Cria explorados como lista vazia de estados explorados
4 Cria solucao como Nenhuma fronteira.enqueue(vertice_inicial)
5 while I == 1 do
6   | if fronteira.vazia() then
7   |   | return solucao
8   | end
9   | vertice = fronteira.dequeue()
10  | if vertice.ehSolucao()  $\wedge$  menor_custo then
11  |   | solucao  $\leftarrow$  vertice
12  | end
13  | if vertice  $\notin$  explorados then
14  |   | explorados.inserir(vertice)
15  |   | fronteira.enqueue(sucessores(vertice))
16  | end
17 end
```

É feita, então, uma busca em largura - BFS (Alg. 1), utilizando o vértice inicial como raiz para encontrar as coleções de custo mínimo. O BFS utiliza uma fila como estrutura de dados de fronteira (nós que serão explorados). Se a fronteira estiver vazia, termina a busca partindo daquele subconjunto. Senão ele retira um subconjunto da fila e verifica se ele, junto com os atuais subconjuntos da coleção, formam uma solução válida e de custo menor que a *solucao*. Caso seja, ela é substituída a *solucao*, caso contrário exploram-se os sucessores do subconjunto corrente, adicionando-os na fila.

O algoritmo ZicaZeroZ (Alg. 2) executa o BFS para todo subconjunto que representa um voluntário. Caso encontre solução, verifica se ela é uma coleção menor do que a atual, substituindo-a. Senão, caso tenha o mesmo custo, verifica se a soma dos índices

Algoritmo 2: Algoritmo ZicaZeroZ proposto

```
1 ZicaZeroZ()
2   melhor_solucao  $\leftarrow$  Nenhuma
3   soma_melhor_solucao  $\leftarrow$  0
4   for  $\forall$  subconjunto  $\in \mathbf{S}$  do
5     | Cria novo_vertice
6     | novo_vertice.estado  $\leftarrow$  subconjunto
7     | nova_solucao  $\leftarrow$  BFS(novo_vertice)
8     | if nova_solucao  $\neq$  Nenhuma then
9       |   | if melhor_solucao == Nenhuma  $\vee$ 
10      |   |   | custo(melhor_solucao)  $>$  custo(nova_solucao) then
11        |   |   |   | melhor_solucao  $\leftarrow$  nova_solucao
12        |   |   |   | soma_melhor_solucao  $\leftarrow$  soma(melhor_solucao)
13        |   |   | end
14        |   | else if custo(melhor_solucao) == custo(nova_solucao)  $\wedge$ 
15          |   |   | soma(nova_solucao)  $<$  soma_melhor_solucao then
16            |   |   |   | melhor_solucao  $\leftarrow$  nova_solucao
17            |   |   |   | soma_melhor_solucao  $\leftarrow$  soma(melhor_solucao)
18        |   | end
19      | end
20  end
```

dos subconjuntos é menor, se sim, substitui a solução atual. Ao final grava resultado em um arquivo de saída.

3. Análise de complexidade (Exercício 3)

3.1. Complexidade de tempo

Considerando-se que o BGS gera uma árvore uniforme com b nós no primeiro nível, cada nó geraria mais b nós no nível inferior e assim em diante até uma profundidade d . Pode-se verificar em [Russell and Norvig 2009] que o número total de nós expandidos seria de:

$$b + b^2 + b^3 + \dots + b^d = O(b^d)$$

Seja, então, n o número de subconjuntos/voluntários do algoritmo. O pior caso (que geraria todos os nós) seria quando todos esses subconjuntos tivessem um relacionamento em **Q** e nenhuma coleção de subconjuntos cobre todos os focos. Neste caso, utilizaria-se n nós no primeiro nível e a altura média de $n/2$ (considerando que o algoritmo verifica estado repetidos).

Isto geraria n nós no nível inferior até a profundidade média de $n/2$. Gerando uma complexidade exponencial: $O(n^{n/2})$

3.2. Complexidade de espaço

Para qualquer tipo de pesquisa em grafo que armazena cada estado explorado em uma lista, a complexidade espaço é sempre um fator b da complexidade de tempo [Russell and Norvig 2009]. Neste caso, haverão $O(n^{n-1})$ nós no conjunto de estados

explorados e $O(n^{n/2})$ nós na fronteira. Assim, a complexidade será dominada pelos nós explorados, trazendo uma complexidade final de $O(n^{n-1})$.

4. Entradas utilizadas (Exercício 5)

Foram utilizados para testes três conjuntos de entradas:

- 1- as disponibilizadas pelo monitor (ZikaZeroZ.zip) para verificar a corretude das respostas.
- 2- as entradas de empate para verificar critérios de empate (quando há soluções do mesmo tamanho).
- 3- entradas geradas por um script desenvolvido para testar o pior caso apresentado anteriormente.

Para gerar uma entrada através do script basta utilizar o comando:

```
python3 input-gen.py NOME_DO_ARQUIVO NUMERO_DE_VOLUNTARIOS
```

Esse script sempre gera o NUMERO_DE_VOLUNTARIOS + 1 como focos e não tem solução.

5. Testes (Exercícios 5 e 6)

Foram realizados testes na máquina do DCC Araguaia.

As saídas disponibilizadas e de empate obtiveram resultados coerentes com o esperado.

n	Tempo (segundos)	# expansões de nó	
		real	teórico
2	0.000	6	2
3	0.001	33	5
4	0.001	124	16
5	0.006	395	56
6	0.017	1146	216
7	0.046	3129	907
8	0.116	8184	4096
9	0.300	20727	19683
10	0.864	51190	100000
11	2.885	123893	534146
12	9.886	294900	2985984
13	35.676	692211	17403307

Figura 2. Resultados obtidos de execuções das entradas do script de tamanho n=2 até n=13

Nas entradas do script, foi executado três vezes o algoritmo para verificar os tempos de execução para entradas de tamanho $n = 2$ até $n = 13$ (que rodaram em menos de 1 minuto cada execução). Assim, foi obtido o tempo médio do processo e o número de nós expandidos (Tab. 2). Colocou-se, também, na tabela valores teóricos de nós expandidos, segundo a análise de complexidade apresentada (que será comparada adiante).

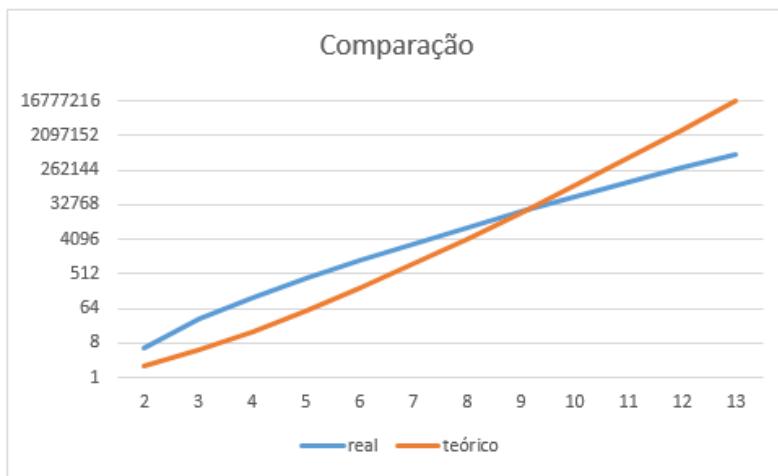


Figura 3. Comparativo entre o número de nós expandidos real e teórico em escala logarítmica (base 2)

Verifica-se no gráfico em escala logarítmica (Fig. 3) de que tanto o número de nós expandidos real quanto teórico são retas, portanto a complexidade é exponencial como previsto. Outra constatação é a de que com $n=8$, a função de complexidade prevista passa a ser um limite assintótico superior para a complexidade do algoritmo, o que é coerente com a notação de pior caso apresentada.

6. Conclusão (Exercício 6)

Foi realizada com sucesso a implementação de um algoritmo para resolver o problema ZicaZeroZ. Os resultados teóricos e práticos de execução foram condizentes com o especificado.

Seria necessária uma avaliação probabilística mais apurada para definir um limite assintótico firme para o algoritmo utilizado. Implementações mais eficientes utilizando heurísticas e algoritmos aproximados são possíveis, mas não estavam no escopo do trabalho em questão.

Referências

Russell, S. and Norvig, P. (2009). *Artificial intelligence: a modern approach*. Prentice Hall, 3rd edition.

Vazirani, V. V. (2001). *Approximation algorithms*. Springer Science & Business Media.

Relatório do Trabalho Prático de Grafos - PAA

Victor Silva Rodrigues
victor.rodrigues@dcc.ufmg.br

2016/1

Exercício 1 - Modelagem

A entrada do problema é apresentada de forma que a modelagem empregando grafos seja bem natural. Deste modo, a modelagem do problema foi feita representando o conjunto \mathbb{V} de voluntários como os vértices do grafo \mathcal{G} . Os laços de amizade entre os voluntários são representados como as arestas que ligam os vértices do grafo, de modo que $\mathbb{E} = \{(u, v) : u, v \in \mathbb{V}\}$ se u e v são amigos.

Além disso, cada vértice possui um subconjunto $\mathcal{F}(v) \subseteq \mathbb{F}$, assumindo-se de tamanho $\in \{1..r\}$, que representa os focos aos quais ele tem acesso.

Uma solução na qual $\bigcup_{v \in \mathbb{V}'} (\mathcal{F}(v)) = \mathbb{F}$, para um subgrafo induzido por $\mathbb{V}' \subseteq \mathbb{V}$ em \mathcal{G} em que \mathbb{V}' é conexo é uma solução para o problema em questão.

Exercício 2 - Algoritmos

A seguir uma listagem dos algoritmos. De modo a otimizar a complexidade dos algoritmos, a implementação da estrutura de dados Grafo foi feita utilizando listas de adjacências (cada vértice de \mathcal{G} é uma entrada em uma tabela hash, e cada vértice possui uma lista com seus vizinhos). Além disso, cada vértice contém uma tabela hash contendo os focos aos quais aquele voluntário tem acesso.

GENERATE-COMBINATIONS(\mathbb{V})

```
1: for  $k \in \{1..|\mathbb{V}|\}$  do
2:   yield each of the  $\binom{|\mathbb{V}|}{k}$  combinations for  $\mathbb{V}$ 
3: end for
```

GENERATE-SUBGRAPH(\mathcal{G}, \mathbb{V}')

```
1:  $\mathcal{G}' \leftarrow (\mathbb{V} = \emptyset, \mathbb{E} = \emptyset)$ 
2: for  $v \in \mathbb{V}'$  do
3:    $\mathcal{G}'(\mathbb{V}) \leftarrow \mathcal{G}'(\mathbb{V}) \cup v$ 
4: end for
5: for  $v \in \mathcal{G}'(\mathbb{V})$  do
6:   for  $u \in \mathcal{G}.adj[v]$  do
7:     if  $u \in \mathcal{G}'(\mathbb{V})$  then
8:        $\mathcal{G}'(\mathbb{E}) \leftarrow \mathcal{G}'(\mathbb{E}) \cup (u, v)$ 
9:     end if
10:   end for
11: end for
```

INITIALIZE(\mathcal{G})

```
1: for  $v \in \mathcal{G}(\mathbb{V})$  do
2:   color[v] = WHITE
3: end for
```

FIND-COMPLETE-COVERAGE(\mathcal{G}, \mathbb{F})

```
1: for each  $\mathbb{V}' \in$  GENERATE-COMBINATIONS( $\mathcal{G}(\mathbb{V})$ ) do
2:    $\mathcal{G}' \leftarrow$  GENERATE-SUBGRAPH( $\mathcal{G}, \mathbb{V}'$ )
3:   INITIALIZE( $\mathcal{G}'$ )
```

```

4:    $f \leftarrow \emptyset$ 
5:    $\text{DFS}(\mathcal{G}', \mathcal{G}'(\mathbb{V})_0, f)$ 
6:   connected  $\leftarrow$  True
7:   for  $v \in \mathbb{V}'$  do
8:     if  $\text{color}[v] \neq \text{BLACK}$  then
9:       connected  $\leftarrow$  False
10:    end if
11:   end for
12:   if connected and  $f = \mathbb{F}$  then
13:     return  $\mathbb{V}'$ 
14:   end if
15: end for

```

```

 $\text{DFS}(\mathcal{G}, \text{root}, f)$ 
1:  $\text{color}[\text{root}] = \text{GREY}$ 
2:  $f \leftarrow f \cup \mathcal{F}(\text{root})$ 
3: for  $u \in \mathcal{G}.\text{adj}[\text{root}]$  do
4:   if  $\text{color}[u] = \text{WHITE}$  then
5:      $\text{DFS}(\mathcal{G}, u, f)$ 
6:   end if
7: end for
8:  $\text{color}[\text{root}] = \text{BLACK}$ 

```

Exercício 3 - Análise de Complexidade

Segundo a sequência na qual os algoritmos foram apresentados, serão apresentados os seus custos em termos de complexidade assintótica.

Generate-Combinations

Isolando a análise na geração de cada combinação de k vértices: para cada uma das $\binom{|\mathbb{V}|}{k}$ combinações, existem $\binom{|\mathbb{V}|}{k}$ operações de incremento (no total) e k operações de rearranjo com custo agregado $\mathcal{O}(k^2)$. Um exemplo de incremento ocorre quando o conjunto de vértices a serem incluídos em \mathbb{V}' (com $|\mathbb{V}| = 10, k = 3$, por exemplo), é o seguinte: $[2, 5, 7] \rightarrow [2, 5, 8]$. Um exemplo de rearrajno ocorre, por exemplo quando $[2, 5, 10] \rightarrow [2, 6, 7]$, ou $[2, 9, 10] \rightarrow [3, 4, 5]$ assumindo que o algoritmo sempre gera todas as $\binom{|\mathbb{V}|}{k}$ combinações em ordem crescente.

Deste modo, o custo total de gerar as $\binom{|\mathbb{V}|}{k}$ combinações é $\mathcal{O}(k^2 + \binom{|\mathbb{V}|}{k})$.

Como são geradas, no total, $\sum_{k=1}^{|\mathbb{V}|} \binom{|\mathbb{V}|}{k} = 2^{|\mathbb{V}|} - 1$ combinações a um custo total $\mathcal{O}\left(\sum_{k=1}^{|\mathbb{V}|} (k^2 + \binom{|\mathbb{V}|}{k})\right)$,

o custo agregado de todas as combinações geradas pelo GENERATE-COMBINATIONS é, então, $\mathcal{O}(2^{|\mathbb{V}|} + |\mathbb{V}|^3) = \mathcal{O}(2^{|\mathbb{V}|})$

A cada geração, um array com os k elementos gerados é mantida na memória. O custo de complexidade de memória deste método é, então, $\mathcal{O}(|\mathbb{V}|)$

Generate-Subgraph

Na linha 1, um grafo vazio é inicializado com uma operação de custo $\mathcal{O}(1)$. A linha 2 extrai um vértice v de \mathbb{V}' , que é representado como um array. Cada extração possui custo $\mathcal{O}(1)$. Na linha 3, a união é feita adicionando v a uma tabela hash, a custo $\mathcal{O}(1)$. Nas linhas 5 e 6, cada aresta incidente a um vértice em \mathcal{G}' é visitada uma vez, e adicionada a \mathcal{G}' na linha 8 com custo $\mathcal{O}(1)$, resultando em um custo agregado de $\mathcal{O}(|\mathbb{V}'| + |\mathbb{E}|) = \mathcal{O}(|\mathbb{V}| + |\mathbb{E}|)$.

O complexidade do custo de espaço também é $\mathcal{O}(|\mathbb{V}| + |\mathbb{E}|)$, que é o custo de se armazenar a representação do subgrafo gerado em listas de adjacencias.

Initialize

Trivialmente, a complexidade assintótica do custo de tempo e espaço desta função é $\mathcal{O}(|\mathcal{G}(\mathbb{V})|) = \mathcal{O}(|\mathbb{V}|)$

DFS

Conforme o usual para DFSSs, o corpo da função é executado um total agregado de $\mathcal{O}(|\mathbb{V}| + |\mathbb{E}|)$ vezes. Isto pode ser provado observando que cada vértice só é acessado uma vez quando sua cor é branca. Assim que é acessado, ele tem sua cor imediatamente modificada para cinza. Quando todos seus vizinhos são explorados, sua cor é modificada para preta: contribuindo com $\mathcal{O}(|\mathbb{V}|)$ ao custo total. Para cada vértice, as arestas adjacentes são exploradas. Como cada vértice só é acessado uma vez, cada aresta só pode ser acessada no máximo 2 vezes durante a execução da DFS (grafo não-direcionado): contribuindo em $\mathcal{O}(|\mathbb{E}|)$ ao custo total. Ignorando então as linhas 3-7, temos as operações nas linhas 1, 2 e 8, que ocorrem $\mathcal{O}(|\mathbb{V}| + |\mathbb{E}|)$ vezes cada. O custo da operação nas linhas 1 e 8 é $\mathcal{O}(1)$. O custo da operação da linha 2 é $|\mathbb{F}|$, pois cada voluntário pode acessar até $|\mathbb{F}|$ focos. f é uma tabela hash, portanto a união entre f e $\mathcal{F}(\text{root})$ tem custo $\mathcal{O}(|\mathbb{F}|)$.

O custo total de execução da DFS é, então, $\mathcal{O}(|\mathbb{F}|(|\mathbb{V}| + |\mathbb{E}|))$

O custo de memória é $\mathcal{O}(|\mathbb{V}|)$ para armazenar as cores de cada vértice e a pilha de execução, e $\mathcal{O}(|\mathbb{F}|)$ para armazenar f , totalizando em $\mathcal{O}(|\mathbb{V}| + |\mathbb{F}|)$

Find-Complete-Coverage

Algumas linhas possuem operações cujo custo foi computado neste exercício. Portanto, diretamente desses custos, para cada combinação de vértices, temos que:

O custo da linha 2 é $\mathcal{O}(|\mathbb{V}| + |\mathbb{E}|)$ (memória: $\mathcal{O}(|\mathbb{V}| + |\mathbb{E}|)$)

O custo da linha 3 é $\mathcal{O}(|\mathbb{V}|)$ (memória: $\mathcal{O}(|\mathbb{V}|)$)

O custo da linha 5 é $\mathcal{O}(|\mathbb{F}|(|\mathbb{V}| + |\mathbb{E}|))$ (memória: $\mathcal{O}(|\mathbb{V}| + |\mathbb{F}|)$).

Além disso: O custo das linhas 4 e 6 é, trivialmente $\mathcal{O}(1)$ (memória $\mathcal{O}(1)$).

A linha 9 é executada $\mathcal{O}(|\mathbb{V}|)$ vezes a custo $\mathcal{O}(1)$, totalizando em $\mathcal{O}(|\mathbb{V}|)$ (memória $\mathcal{O}(1)$).

A linha 12 pode ser feita em $\mathcal{O}(1)$ se forem mantidos os tamanho de \mathbb{F} e f em memória.

Somando todos os custos e considerando apenas o dominante, temos, para cada combinação de vértices, um custo assintótico de tempo de $\mathcal{O}(|\mathbb{F}|(|\mathbb{V}| + |\mathbb{E}|))$ e de espaço $\mathcal{O}(|\mathbb{F}| + |\mathbb{V}| + |\mathbb{E}|)$

Como após cada iteração o subgrafo gerado \mathcal{G}' não é mais utilizado, ele é removido da memória. Da mesma forma, f é removido. Deste modo, o custo de memória permanece constante em $\mathcal{O}(|\mathbb{F}| + |\mathbb{V}| + |\mathbb{E}|)$ durante toda a execução.

Como existem $2^{|\mathbb{V}|}$ combinações de vértices, o custo total de execução é $\mathcal{O}(2^{|\mathbb{V}|}|\mathbb{F}|(|\mathbb{V}| + |\mathbb{E}|))$.

Somando o custo agregado da linha 1 (ou seja, o custo agregado de GENERATE-COMBINATIONS) ao custo total, temos que o custo de execução é, finalmente, $\mathcal{O}(2^{|\mathbb{V}|}|\mathbb{F}|(|\mathbb{V}| + |\mathbb{E}|)) + \mathcal{O}(2^{|\mathbb{V}|}) = \mathcal{O}(2^{|\mathbb{V}|}|\mathbb{F}|(|\mathbb{V}| + |\mathbb{E}|))$, enquanto que o custo de memória é $\mathcal{O}(|\mathbb{F}| + |\mathbb{V}| + |\mathbb{E}|) + \mathcal{O}(|\mathbb{V}|) = \mathcal{O}(|\mathbb{F}| + |\mathbb{V}| + |\mathbb{E}|)$.

Exercício 5 - Execução

Além dos casos de teste fornecidos pelo monitor no fórum de discussão, para os quais o algoritmo executou corretamente, foi implementado um gerador de casos de teste.

O gerador foi projetado para atender aos seguintes requisitos:

- Ser capaz de gerar instâncias e suas respectivas soluções.
- Ser capaz de gerar instâncias cujo tamanho da solução seja um parâmetro k ; exista pelo menos uma solução ótima de tamanho k ; e não exista solução ótima menor do que k .
- n, m, r também sejam parâmetros de entrada do gerador, que tenta gerar grafos respeitando esses tamanhos.

A maneira mais natural de criar o gerador foi a seguinte:

1. Escolher, aleatoriamente, k dentre os n vértices.
2. Criar um grafo linha com os k vértices escolhidos.
3. No primeiro vértice do grafo linha, colocar os $r - 1$ primeiros focos ($F_i : i \in \{1..r - 1\}$) como focos aos quais o vértice tem acesso.
4. No último vértice do grafo linha, colocar último foco F_r como foco a qual ele tem acesso.
5. Nos demais vértices, colocar aleatoriamente focos, exceto o primeiro e o último focos ($F_i : i \in \{2..r - 1\}$), como focos aos quais eles têm acesso.
6. Inserir arestas aleatoriamente entre os vértices que não pertencem ao grafo linha, podendo incluir somente o último vértice do grafo linha, garantindo que todas as arestas do grafo linha continuem sendo pontes.

A prova de que o gerador funciona segue dos fatos de que: qualquer solução precisa incluir o primeiro vértice do grafo linha (pois é o único que contém o foco F_1) e o último vértice do grafo linha (pois é o único que contém o foco F_r). O grafo linha é conexo, pela forma como foi construído. Além disso, só existe um caminho entre o primeiro e o último vértices do grafo linha. Como a união entre os focos do primeiro e último vértices cobre todo o conjunto de focos, a solução cujos vértices selecionados são os vértices do grafo linha é válida. Ela é mínima pois não existe maneira de chegar do primeiro ao último vértice sem passar por todos os demais do grafo linha. É máxima pois qualquer outra solução válida inclui necessariamente o grafo linha, e é maior em pelo menos um vértice. ■

A seguir, serão apresentados alguns exemplos de instâncias geradas e, em seguida, os tempos de execução experimentais serão reportados.

Exemplos de Instâncias Geradas

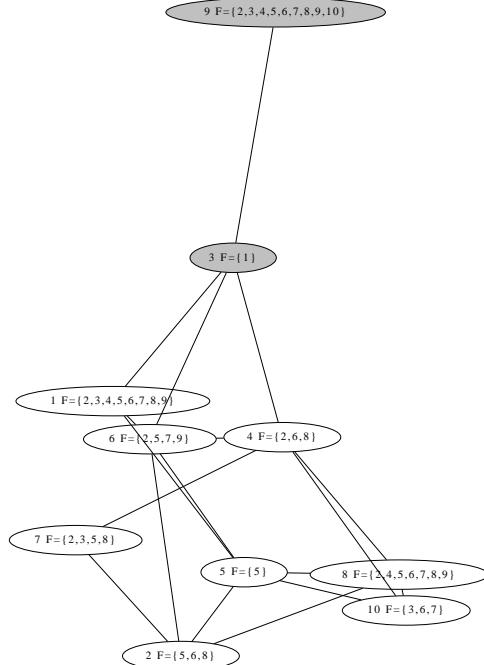


Figure 1: Exemplo de instância gerada para $n = 10, m = 18, r = 10, k = 2$

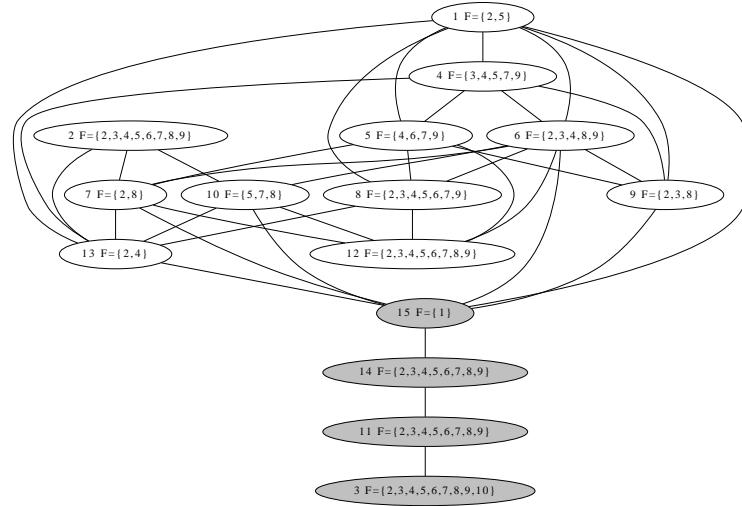


Figure 2: Exemplo de instância gerada para $n = 15, m = 37, r = 10, k = 4$

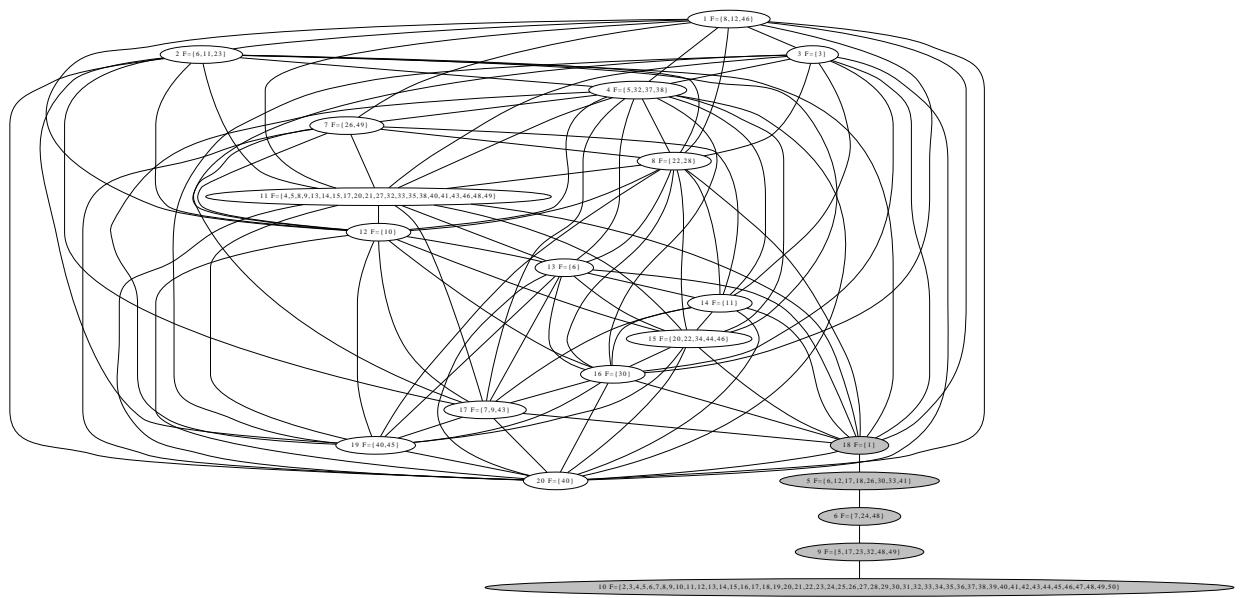


Figure 3: Exemplo de instância gerada para $n = 20, m = 95, r = 50, k = 5$

Tempos de Execução

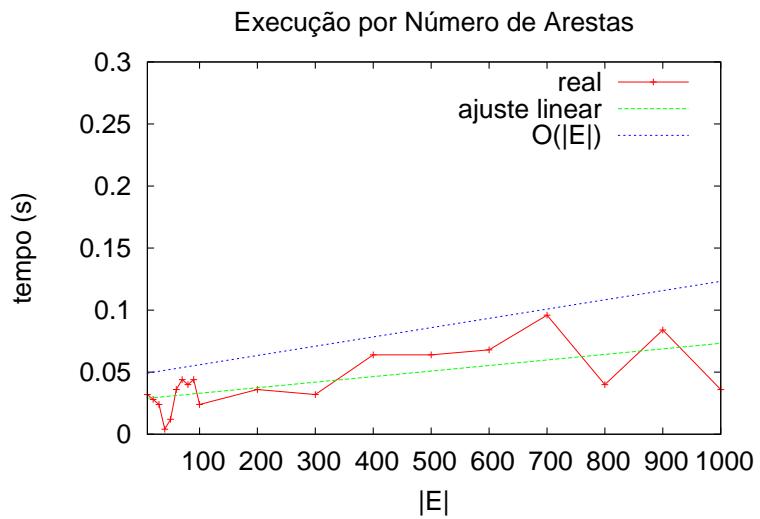


Figure 4: Gráfico de tempos de execução para diferentes quantidades de arestas, com $|V|, |F|$ fixos.

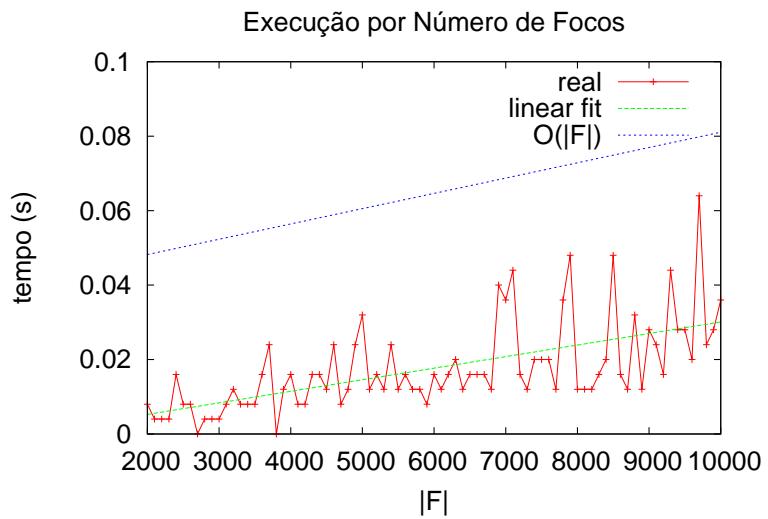


Figure 5: Gráfico de tempos de execução para diferentes quantidades de focos, com $|V|, |E|$ fixos.

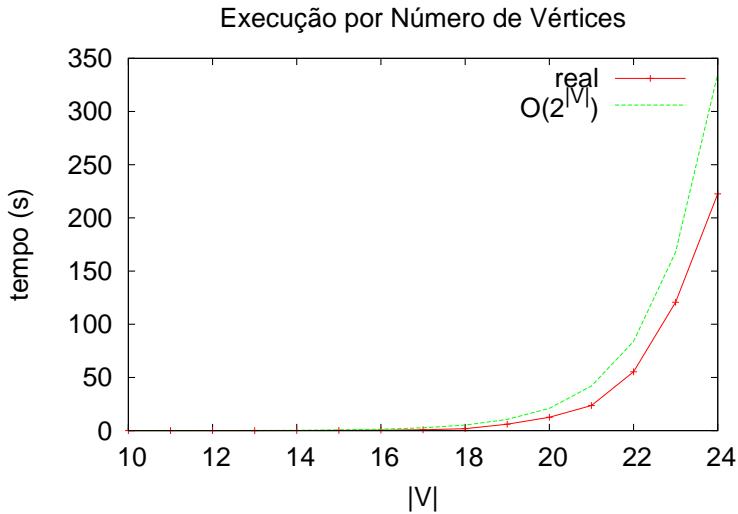


Figure 6: Gráfico de tempos de execução para diferentes quantidades de vértices, com $|\mathbb{E}|, |\mathbb{F}|$ fixos.

Exercício 6 - Comparação dos Custos com os Tempos de Execução

Conforme pode-se observar no gráfico da figura 4, o ajuste linear sobre os pontos da execução resultaram em uma inclinação positiva. Para os tamanhos de entrada testados, variando de 100 a 1000, existe uma evidência forte de que a complexidade do custo do algoritmo seja linear com relação a $|\mathbb{E}|$, fixados os demais parâmetros. O mesmo pode-se dizer sobre $|\mathbb{F}|$, observando a figura 5.

Ao analisar o custo de tempo de $|\mathbb{V}|$ pela figura 6, para tamanhos de entrada de 10 a 24, percebe-se que a curva sobe rapidamente de milissegundos à casa dos 3 minutos. Foi plotada uma curva exponencial para limitar assintoticamente por cima os pontos do tempo de execução medido. O limite assintótico de $\mathcal{O}(2^{|V|})$ é válido para o intervalo estudado, fixados os demais parâmetros, levando a crer que os tempos reais de execução corroboram com a análise assintótica teórica apresentada neste relatório.



Tiago Pimentel Martins da Silva
Projeto e Análise de Algoritmos
Pr. Sebastián Urrutia

Trabalho Prático 01- Grafos

1. Modelagem do Problema Utilizando Grafos

Para esse problema temos:

- Um grafo não direcionado $G(V, A)$;
- Um conjunto F de tamanho r com os focos de reprodução do mosquito;
- Uma relação $R(v): V \rightarrow F$, que indica a quais focos cada voluntário tem acesso.

Define-se:

$$V = \{v_0, v_1, \dots, v_{|V|}\}$$

E queremos encontrar o menor subconjunto $V' \subseteq V$, tal que o grafo induzido por V' em G seja conexo e para todo $f \in F$, exista um $v \in V'$, tal que $f \in R(v)$. Numerando esses objetivos:

1. $V' \subseteq V$;
2. $G'(V', A')$ seja conexo;
3. $\forall f \in F, \exists v \in V': f \in R(v)$;
4. $V' = \arg \min_{\bar{V} \subseteq V} (|\bar{V}|)$.

Para resolver esse problema, pode-se criar um novo conjunto de vértices P , tal que:

$$P = \{s = (\emptyset)\} \cup \bigcup_{k=1}^{|V|} \binom{V}{k},$$

Sendo $\binom{V}{k}$ um conjunto com todas as combinações de vértices $v \in V$, tomados k a k :

$$P_k = \binom{V}{k} = \{p_k = (v_{l[1]}, \dots, v_{l[k]}) | \forall l \in C(|V|, k)\}.$$

$$P_0 = \binom{V}{0} = \{s = (\emptyset)\}$$

$C(|V|, k)$ é um conjunto com todas as combinações dos números 1 a $|V|$ tomados k a k elementos. Note que cada elemento $p_k \in P_k \subset P$ é, por si só, um conjunto não ordenado de k elementos $v_i \in V$. Cada conjunto P_k terá $\binom{|V|}{k}$ elementos e o conjunto P de vértices terá tamanho $|P| = 2^{|V|}$ (veja prova em Anexo 1).

$$|P_k| = \binom{|V|}{k} = \frac{|V|!}{(|V| - k)! \cdot k!}$$

$$|P| = \sum_{k=0}^{|V|} |P_k| = 2^{|V|}$$

Um novo conjunto de arestas direcionadas E também é criado, derivado do conjunto de relações de amizades A .

$$E = \left\{ (p_k, p_{k+1}) \middle| \begin{array}{l} \forall v \in p_k: v \in p_{k+1} \\ \exists v_i \in p_k, \exists v_j \in p_{k+1}: v_j \notin p_k, (v_i, v_j) \in A, \\ p_k \in P_k, p_{k+1} \in P_{k+1}, \\ 0 < k \leq |V| - 1, \end{array} \right\} \cup \{(s, p_1) \mid p_1 \in P_1\}$$

Esse novo grafo $G'(P, E)$ é direcionado e em camadas, com vértices em cada camada k tendo apenas arestas de saída para as respectivas camadas $k+1$. Cada vértice $p_k \in P_k$ pode ter até $|V| - k$ arestas de saída, então o total de arestas será (veja prova em Anexo 2):

$$|E| \leq |V| \cdot 2^{|V|-1}$$

$$|E| \in O(|V| \cdot 2^{|V|})$$

A última coisa necessária para resolver o problema é um conjunto de vértices objetivo $T \subseteq P$:

$$T = \{p \in P \mid \forall f \in F, \exists v \in p: f \in R(v)\}$$

Esse conjunto T representa todos os possíveis conjuntos de voluntários que satisfazem a terceira regra a ser respeitada, que indica que todo foco deve ser acessível por ao menos um voluntário. Veja que o conjunto T pode ter tamanho máximo P , se incluir todos os elementos desse conjunto. Então $|T| \in O(|P|) = O(2^{|V|})$.

2. Algoritmo

Com o grafo $G'(P, E)$, descrito na sessão 1, utiliza-se uma busca em largura, *Breadth First Search* (BFS), partindo de $s = \{\emptyset\}$, único elemento de P_0 , até um elemento em T ser encontrado. Essa busca retornará, se existir, um subgrupo de voluntários $V' \in V$ com tamanho (número de elementos) mínimo (A corretude do algoritmo é provada ao fim dessa sessão).

Considerando que a complexidade espacial do BFS é $O(b^d)$, sendo:

- b o *Branching Factor* do grafo, ou seja, o número máximo de arestas saindo de cada vértice, para esse grafo $b = |V|$;
- d a profundidade do objetivo, no caso, a profundidade será o número mínimo de voluntários que satisfaçam os objetivos do problema, $|V'|$.

A complexidade espacial seria, então, $O(|V|^{|V'|})$, uma função que cresce muito rapidamente. Utiliza-se então o algoritmo *Iterative Deepening Depth First Search* (IDDFS). Esse algoritmo utiliza, como o nome indica, o algoritmo *Depth First Search* (DFS), mas o limitando a profundidades máximas iterativamente. Ao limitar as profundidades do DFS, o IDDFS mantém a propriedade do BFS de achar caminhos mínimos até um nó objetivo, mas tem complexidade espacial igual à do DFS, $O(d) = O(|V'|)$. Note que, como cada vértice tem, no máximo, uma aresta de entrada, não é necessário se manter uma gravação de que vértices já foram visitados durante a busca.

SimpleExactZikaZeroZ(graph, s)

```

1 For depth = 1:∞:
2   volunteers = LimitedDFS(graph, s, depth)
3   If volunteers ≠ null:
4     Return volunteers

```

SimpleLimitedDFS (graph, node, depth)

```

1 If depth == 0 && node in goal:
2   Return node
3 Elsif depth > 0:
4   For child = node.adj():
5     goal = LimitedDFS(graph, child, depth-1)
6     If goal ≠ null:
7       Return goal
8   Return null

```

Outro custo a ser considerado é o do próprio grafo. Como discutido na sessão 1, $|P| = 2^{|V|}$ e $|E| \in O(|V| \cdot 2^{|V|})$, então $|P| + |E| \in O(|V| \cdot 2^{|V|})$. Para reduzir esses custos, o grafo $G'(P, E)$ é gerado dinamicamente e só o grafo original $G(V, A)$ é guardado em memória, com complexidade clara:

$$|V| + |A| \in O(|V| + |A|)$$

$$|V| + |A| \leq |V| + |V|^2 \in O(|V|^2)$$

É necessária então uma função para encontrar todos os nós adjacentes a um dado nó $p \in P$.

FindAdjNodes(node)

```

1 If node is empty: //Means it's the s node
2   children = 1:num_volunteers
3 Else:
4   children = []
5   For original_node in node:
6     For child in original_node.adj():
7       If child not in node && child not in children:
8         children.add(child)
9
10 adj_nodes = []
11 For child in children:
12   adj_nodes.append((node + child, child.index))
13
14 Return adj_nodes

```

Lembre que cada nó $p \in P$ do grafo $G'(P, E)$ é um conjunto de nós do grafo original, $p = (v_{l[1]}, \dots, v_{l[k]})$. Esse algoritmo, então, para cada nó $v \in p \subseteq V$, acha todos os nós $w \in V$ adjacentes a ele que não estão em p ($w \notin p$) e cria um conjunto $P' \subset P$ que seja a união de p com $\{w\}$:

$$P' = \{p \cup \{w\} \mid w \notin p, \exists v \in p: (v, w) \in A\}$$

Além disso, o próprio conjunto de objetivos T é checado dinamicamente para não ser alocado em memória. Lembre que $|T| \in O(2^{|V|})$, que é outra função que cresce rapidamente.

CheckIfNodeIsGoal(node)

```

1 num_focuses = Config.r
2
3 For original_node in node:
4   For focus in original_node.mosquito_focus():
5     If focus not in focus_set:
6       focus_set.add(focus)
7     If (length(focus_set) == num_focuses)
8       Return true
9
10 Return false

```

O algoritmo completo para resolver o problema do ZikaZeroZ é:

ExactZikaZeroZ(root, num_volunteers)

```

1 For depth = 1:num_volunteers:
2   volunteers = LimitedDFS(root, depth, [])
3   If volunteers ≠ null:
4     Return volunteers

```

LimitedDFS (node, depth, explored_nodes)

```

1 If depth == 0 && CheckIfNodeIsGoal(node):
2   Return node
3 Elsif depth > 0:
4   For (child, new_vertex_index) in FindAdjNodes(node):
5     If not explored_nodes[new_vertex_index]:
6       explored_nodes[new_vertex_index] = 1
7       goal = LimitedDFS(child, depth-1, explored_nodes)
8       If goal ≠ null:
9         Return goal
10 Return null

```

Corretude do Algoritmo

Nessa seção é argumentado porque o algoritmo BFS (e o IDDFS, por consequência) encontram uma solução ótima para esse problema, atendendo os quatro objetivos, repetidos aqui para clarificar o problema:

1. $V' \subseteq V$;
2. $G'(V', A')$ seja conexo;

3. $\forall f \in F, \exists v \in V': f \in R(v);$
4. $V' = \arg \min_{\bar{V} \subseteq V} (|\bar{V}|).$

Primeira condição:

Pela definição de p :

$$p = (v_{l[1]}, \dots, v_{l[k]})$$

Então todo $p \in P$, se escolhido como V' , satisfaz a **primeira condição**.

$$p = V' \subseteq V$$

Terceira condição:

O algoritmo *LimitedDFS* é um algoritmo recursivo com caso base $depth = 0$, para o qual ele retorna:

$$\begin{cases} node, & \text{if } \text{CheckIfNodeIsGoal}(node) \\ \text{null}, & \text{else} \end{cases}$$

Os únicos nós retornados por esse algoritmo serão, então, os que satisfizerem a função *CheckIfNodeIsGoal*. Essa função checa se um nó $p \in P$ satisfaz a condição de que, para todo $f \in F$ exista ao menos um $v \in p$, tal que $f \in R(v)$. Reescrevendo essa condição temos:

$$\forall f \in F, \exists v \in p: f \in R(v)$$

Que é exatamente a **terceira condição**, provando então que o algoritmo *LimitedDFS* retorna apenas nós que satisfazem essa condição.

Segunda condição:

Prova-se agora que o algoritmo retornará apenas conjuntos $p = V'$ que satisfaçam a **segunda condição**. Para provar isso, utiliza-se um argumento induutivo, afirmando que ele explorará um elemento p se, e somente se, $p \in P' \subseteq P$ induz no grafo original G um grafo conexo.

A condição inicial, para $depth = 0$, o algoritmo explora $s = \{\emptyset\}$, que é por si conexo. Na segunda etapa, $depth = 1$, serão explorados todos os elementos $P_1 = \{(v_i) \mid 0 \leq i < |V|\}$, que são todos os elementos conexos com tamanho $|p| = 1$.

Dado uma profundidade $depth = k$, assumimos que todos os elementos $p_k \in P'_k \subseteq P_k$ explorados são conexos. Na etapa $depth = k + 1$, então, exploraremos os elementos $p_{k+1} \in P'_{k+1} \subseteq P_{k+1}$, tais que $\exists p_k \in P'_k: (p_k, p_{k+1}) \in E$. Podemos dividir o conjunto de arestas em:

$$E_0 = \{(s, p_1) \mid p_1 \in P_1\}$$

$$E_k = \left\{ (p_k, p_{k+1}) \left| \begin{array}{c} \forall v \in p_k: v \in p_{k+1}, \\ \exists v_i \in p_k, \exists v_j \in p_{k+1}: v_j \notin p_k, (v_i, v_j) \in A, \\ p_k \in P_k, p_{k+1} \in P_{k+1} \end{array} \right. \right\}$$

Para existir uma aresta $(p_k, p_{k+1}) \in E$, significa que:

$$\exists v \in p_k, \exists w \in p_{k+1}: w \notin p_k, (v, w) \in A$$

e:

$$\forall v \in p_k: v \in p_{k+1}$$

$p_{k+1} \in P_{k+1}$ tem, então, todos os elementos de $p_k \in P_k$ mais $w = v_j$.

$$p_{k+1} = p_k \cup (w)$$

Como, pela definição das arestas: $\exists v_i \in p_k: (v_i, w) \in A$. O elemento w é conexo a, pelo menos, um elemento de p_k . Como p_k é conexo, $p_k \cup (w)$ é também conexo, o que implica em p_{k+1} ser conexo.

Então, todo elemento $p \in P' \subseteq P$ explorado pelo algoritmo BFS (ou IDDFS) é conexo.

Agora, provando que todo elemento conexo é explorado, assume-se que um elemento $p_{k+1} \in P_{k+1}$ é conexo. E assumindo, pela prova iterativa, que todo p_k conexo é explorado, pode-se reduzir p_{k+1} :

$$p_{k+1} = p_k \cup (w)$$

Sendo w um vértice qualquer dentro de p_{k+1} tal que p_k seja um elemento conexo. Isso pode ser feito, sem perda de generalidade, pois existem ao menos dois nós em todo conjunto conexo que podem ser removidos mantendo-se a conexão entre o restante. Por essa definição tem-se que:

$$\forall v \in p_k: v \in p_{k+1}$$

$$w \notin p_k$$

Como p_{k+1} é conexo, sabe-se que:

$$\exists v_i \in p_k: (v_i, w) \in A$$

Com isso, pode-se concluir que existe uma aresta $(p_k, p_{k+1}) \in E$ e, como p_k é explorado, p_{k+1} também será.

Quarta condição:

Pelo modo como o grafo é construído, em camadas, todo elemento atingível pelo elemento $s \in P_0$, terá uma distância igual ao número de camadas entre eles. Um elemento $p_k \in P_k$ na camada k , por exemplo, se for alcançável por s , terá distância k .

Note que o tamanho dos elementos $p_k \in P_k$ é também, pela própria definição de P_k :

$$|p_k| = k.$$

Então nós menores estarão sempre mais próximos de s , caso sejam alcançáveis, do que nós maiores. Como os algoritmos BFS e IDDFS exploram um grafo em ordem de proximidade de um elemento à raiz, um elemento menor sempre será explorado antes de um maior. Então caso haja dois elementos p_k e p_j , $k < j$, que satisfaçam as três primeiras condições do problema, o algoritmo explorará primeiramente o elemento p_k , o retornando e parando sua execução. Isso satisfaz a **quarta condição**.

3. Análise de Complexidades

Nessa seção, serão primeiramente analisadas as complexidades dos algoritmos *CheckIfNodesGoal* e *FindAdjNodes*, e depois serão analisadas as dos algoritmos *LimitedDFS* e *ExactZikaZeroZ*, que dependem deles. Por simplicidade, os algoritmos serão copiados aqui.

CheckIfNodesGoal

CheckIfNodesGoal(node)

```
1 num_focuses = Config.r
2
3 For original_node in node:
4     For focus in original_node.mosquito_focus:
5         If focus not in focus_set:
6             focus_set.add(focus)
7             If (length(focus_set) == num_focuses)
8                 Return true
9
10 Return false
```

Cada nó (*node*) $p \in P$, pode ter tamanho máximo $|p| = |V|$ e ter acesso a no máximo, todos os $r = |F|$ diferentes focos de reprodução do mosquito. Utilizando a estrutura de dados *set* do *Python*, o pior caso para checar se um item existe ou adicionar um novo na lista é $O(\text{len}(\text{set}))$, mas o caso médio é $O(1)$. *len(set)* terá valor máximo r . Essa estrutura tem complexidade $O(1)$ para se obter seu tamanho.

Utilizando um *array* pré-alocado pode-se garantir uma complexidade temporal $O(1)$ tanto para a função de adicionar um item à *array*, quanto para se obter o tamanho dela.

A complexidade temporal desse algoritmo será, então:

- Utilizando *set* – Pior caso: $O(|V| \cdot r^2)$; Caso médio: $O(|V| \cdot r)$
- Utilizando *array* – Pior caso: $O(|V| \cdot r)$; Caso médio: $O(|V| \cdot r)$

A complexidade espacial será dada pela variável *focus_set* e será $O(r)$, que é o tamanho máximo que ela pode obter. As outras variáveis têm complexidade espacial $O(1)$, considerando que *original_node* use ponteiros para representar a lista *mosquito_focus*. Caso *original_node* seja representada por um cópia na memória, ainda assim sua lista interna *mosquito_focus* terá complexidade espacial $O(r)$, não aumentando a complexidade espacial da função.

- Complexidade temporal: $O(|V| \cdot r)$;
- Complexidade espacial: $O(r)$.

FindAdjNodes

FindAdjNodes(node)

```
1 If node is empty: //Means it's the s node
2     children = 1:num_volunteers
```

```

3 Else:
4     children = []
5     For original_node in node:
6         For child in original_node.adj():
7             If child not in node && child not in children:
8                 children.add(child)
9
10 adj_nodes = []
11 For child in children:
12     adj_nodes.append((node + child, child.index))
13
14 Return adj_nodes

```

A linha 2 desse algoritmo terá uma complexidade temporal $O(|V|)$. Realizando uma análise amortecida nas linhas 5 a 8 desse algoritmo, pode-se concluir que o if da linha 7 será executado $O(|A|)$ vezes. Novamente, as checagens da linha 4 variam em complexidade temporal para o caso de se usar um *array* pré-alocado ou um *set* da linguagem *Python*, ela depende da estrutura de dados utilizada. Para um *array* ambas as checagens podem ser feitas em tempo $O(1)$. A linha 8 será executada um máximo de $|V|$ vezes e, usando um *array*, também tem complexidade $O(1)$.

O for da linha 11 será executado um máximo de $|V|$ vezes e a complexidade da linha 12 depende da estrutura de dados utilizada. Como uma copia tem de ser feita do nó original e adicionado a ele o elemento child, para a estrutura *set* tem-se uma complexidade $O(|V|)$.

A complexidade temporal desse algoritmo será, então, $O(|A| + |V|^2) = O(|V|^2)$.

É importante notar que, para grafos fortemente esparsos, o número de crianças de um nó será $b \ll |V|$ e a complexidade do For nas linhas 11 e 12, para esses casos passa a ser $O(|A| + b \cdot |V|)$. Como, para grafos fortemente esparsos conexos $|A| = O(|V|)$, a complexidade temporal dessa função passa a ser $O(b \cdot |V|) \approx O(|V|)$.

A complexidade espacial se dará pela variável *adj_nodes*. Como o máximo de crianças para um nó p_k é $|V| - k$ e o tamanho de um nó p_{k+1} é $k + 1$, o tamanho de *adj_nodes* será $O(|V|^2)$.

- Complexidade temporal: $O(|V|^2)$;
- Complexidade espacial: $O(|V|^2)$.

LimitedDFS

LimitedDFS (node, depth, explored_nodes)

```

1 If depth == 0 && CheckIfNodeIsGoal(node):
2     Return node
3 Elsif depth > 0:
4     For (child, new_vertex_index) in FindAdjNodes(node):
5         If not explored_nodes[new_vertex_index]:
6             explored_nodes[new_vertex_index] = 1
7             goal = LimitedDFS(child, depth-1, explored_nodes)
8             If goal ≠ null:

```

9 Return goal

10 Return null

Esse é um algoritmo recursivo com profundidade máxima igual a $d = |V|$. O algoritmo DFS tem complexidade temporal $O(|P| + |E|)$ e espacial $O(d = |V|)$. A versão limitada em profundidade terá, então, complexidade espacial $O(depth)$ e temporal (computada pelo número de execuções do if na linha 15):

$$times_executed_5 \leq |E_{limit_depth}|$$

$$times_executed_5 \leq \sum_{k=0}^{depth} (|V| - k) \binom{|V|}{k} \leq |V| \cdot 2^{|V|-1}$$

Pela análise da complexidade temporal a partir do número de ramificações da árvore de busca:

$$times_executed_5 \leq b^{depth} \leq |V|^{depth}$$

Sendo b , novamente, o *Branching Factor* do grafo, para esse grafo $\max(b) = |V|$. Como as linhas 15 a 19 tem complexidade temporal $O(1)$, a complexidade dessa parte total será $O(\min(b^{depth}, |V| \cdot 2^{|V|-1}))$, que, no pior caso, é $O(\min(|V|^{depth}, |V| \cdot 2^{|V|-1}))$.

A função *FindAdjNodes*, por outro lado, será executada um máximo de:

$$|P_{limit_depth}| = \sum_{k=0}^{depth} \binom{|V|}{k} \leq 2^{|V|}$$

pois essa função é chamada, **no máximo**, uma vez por nó. Pela análise do número de ramificações da árvore de busca:

$$times_executed_4 \leq b^{depth-1} \leq |V|^{depth-1}$$

Note que para um $G(V, A)$ esparso, o número de nós alcançáveis dentre os P_{limit_depth} é muito baixo e então:

$$times_executed_4 \leq b^{depth-1} \ll |V|^{depth-1}$$

Como $|P| = 2^{|V|}$ e *FindAdjNodes* tem uma complexidade temporal de $O(|V|^2)$, a complexidade dessa parte será $O(|V|^2 \cdot \min(b^{depth}, 2^{|V|}))$, que pode ser escrito também como $O(|V|^2 \cdot \min(|V|^{depth}, 2^{|V|}))$.

A checagem *CheckIfNodeIsGoal* será executada apenas quando $depth = 0$, ou seja, um máximo de $|P_{depth}|$ vezes.

$$|P_{depth}| = \binom{|V|}{depth}$$

$$|P_{depth}| = \frac{|V|!}{(|V| - depth)! \cdot depth!} \leq |V|^{depth}$$

Como a complexidade dessa função é $O(|V| \cdot r)$, a complexidade total dessa parte será $O(|V|^{depth+1} \cdot r)$. Novamente, para um grafo esparsa $G(V, A)$, o número de elementos alcançáveis dentro de P_{depth} seria muito mais baixo:

$$times_executed_1 \leq b^{depth} \ll |V|^{depth}$$

Temos então a complexidade de *LimitedDFS* como:

$$\begin{aligned} & O(\min(b^{depth}, |V| \cdot 2^{|V|-1}) + |V|^2 \cdot \min(b^{depth}, 2^{|V|}) + \min(b^{depth}, |V|^{depth}) \cdot |V| \cdot r) \\ & O(\min(|V|^{depth}, |V| \cdot 2^{|V|-1}) + |V|^2 \cdot \min(|V|^{depth}, 2^{|V|}) + |V|^{depth+1} \cdot r) \\ & O(\min(|V|^{depth+2}, |V|^2 \cdot 2^{|V|}) + |V|^{depth+1} \cdot r) \end{aligned}$$

A complexidade espacial, por outro lado, será dada por d vezes a complexidade de cada execução dessa função. A única variável nessa função que tem complexidade maior que $O(1)$ é *explored_nodes*, que tem complexidade $\Theta(|V|)$. A complexidade espacial dessa função será, então, isso mais as complexidades das funções chamadas por ela, o que resulta em:

$$O(depth \cdot |V| + |V|^2 + r)$$

Como ambos $depth \leq |V|$ e $r \leq |V|$:

$$O(|V|^2)$$

- Complexidade temporal: $O(\min(|V|^{depth+2}, |V|^2 \cdot 2^{|V|}) + |V|^{depth+1} \cdot r)$;
- Complexidade espacial: $O(|V|^2)$.

ExactZikaZeroZ

`ExactZikaZeroZ(root, num_volunteers)`

```

1 For depth = 1:num_volunteers:
2   volunteers = LimitedDFS(root, depth, [])
3   If volunteers ≠ null:
4     Return volunteers

```

Esse algoritmo rodará a linha 2 um número máximo de $|V|$ vezes, gerando um pior caso com complexidade (os termos simplificados da função anterior foram novamente expandidos):

$$\begin{aligned} comp &= \sum_{depth=1}^{|V|} \left(\sum_{k=0}^{depth} (|V| - k) \binom{|V|}{k} + r \cdot |V| \cdot \binom{|V|}{depth} + \sum_{k=0}^{depth} |V|^2 \cdot \binom{|V|}{k} \right) \\ comp &= \sum_{depth=1}^{|V|} \sum_{k=0}^{depth} (|V| - k) \binom{|V|}{k} + \sum_{depth=1}^{|V|} r \cdot |V| \cdot \binom{|V|}{depth} + \sum_{depth=1}^{|V|} \sum_{k=0}^{depth} |V|^2 \cdot \binom{|V|}{k} \end{aligned}$$

Como $|V| - k < |V|^2$:

$$comp \leq r \cdot |V| \cdot (2^{|V|} - 1) + 2 \cdot \sum_{depth=1}^{|V|} \sum_{k=0}^{depth} |V|^2 \cdot \binom{|V|}{k}$$

E como (veja a prova em Anexo 4):

$$\sum_{depth=1}^{|V|} \sum_{k=0}^{depth} \binom{|V|}{k} = (|V| + 2) \cdot 2^{|V|-1} - 1,$$

$$comp \leq r \cdot |V| \cdot 2^{|V|} + 2 \cdot |V|^2 \cdot ((|V| + 2) \cdot 2^{|V|-1} - 1)$$

$$comp \in O(r \cdot |V| \cdot 2^{|V|} + |V|^3 \cdot 2^{|V|})$$

Como $r \leq |V|$:

$$comp \in O(|V|^3 \cdot 2^{|V|})$$

Tem-se então, como complexidade temporal assintótica, para o pior caso $O(|V|^3 \cdot 2^{|V|})$, em que a resposta inclui todos os voluntários, $V' = V$, e se tem de procurar toda a árvore de possibilidades.

Para um problema com resposta em que $x = |V'| < |V|$, tem-se um resultado melhor para a complexidade temporal:

$$comp = \sum_{depth=1}^x \sum_{k=0}^{depth} (|V| - k) \binom{|V|}{k} + \sum_{depth=1}^x r \cdot |V| \cdot \binom{|V|}{depth} + \sum_{depth=1}^x \sum_{k=0}^{depth} |V|^2 \cdot \binom{|V|}{k}$$

Novamente, como $|V| - k < |V|^2$:

$$comp \leq r \cdot |V| \cdot \sum_{depth=1}^x \binom{|V|}{depth} + 2 \cdot |V|^2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k}$$

Note que, para diferentes valores de x temos uma diferença drástica em performance.

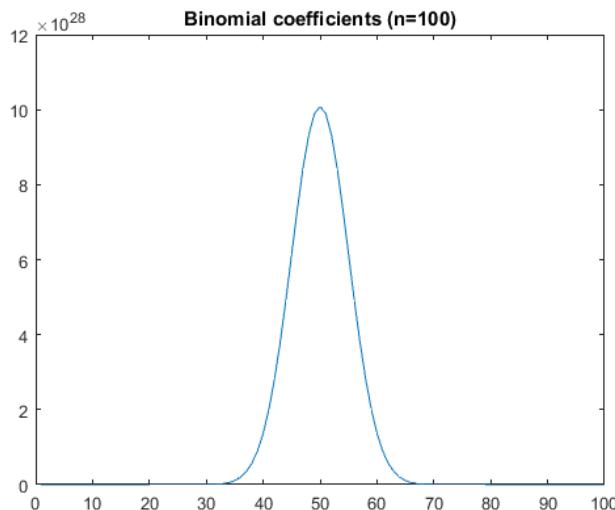


Figura 1 - Coeficientes Binomiais para $n=100$

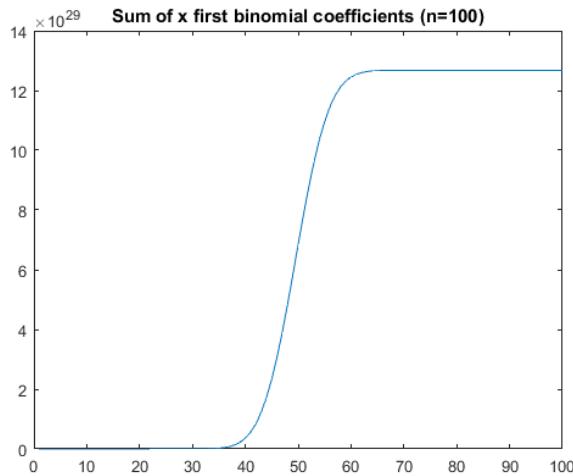


Figura 2 - Soma dos primeiros x coeficientes binomiais para $n=100$

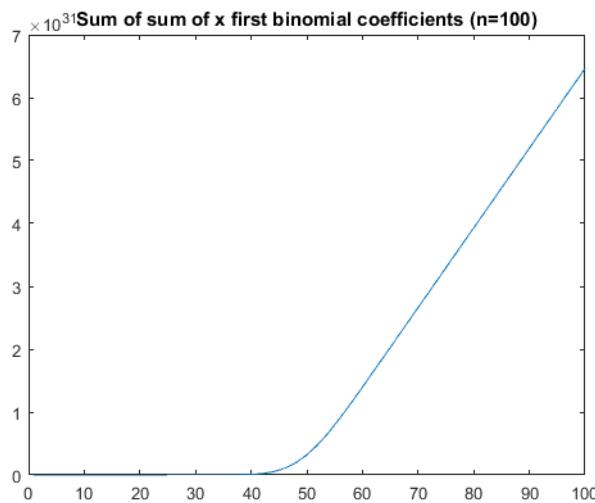


Figura 3 - Soma da soma dos primeiros x coeficientes binomiais para $n=100$

Essa função tem como complexidade espacial a complexidade de LimitedDFS, pois todos seus outros termos tem complexidade $O(1)$. Sua complexidade é, então $O(|V|^2)$.

- Complexidade temporal (sendo $x = |V'|$ o número mínimo de voluntários que satisfaz o problema):

$$O\left(r \cdot |V| \cdot \sum_{depth=1}^x \binom{|V|}{depth} + |V|^2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k}\right)$$

- Complexidade espacial:

$$O(|V|^2)$$

4. Implementação do Algoritmo

O algoritmo foi implementado na linguagem *Python*.

5. Execução de testes

Testes foram executados para variações dos parâmetros:

- $n = |V|$, número de voluntários;
- $m = |A|$, número total de amizades;
- $x = |V'|$, número de voluntários na solução ótima;
- $r = |F|$, número de focos de reprodução do mosquito.

Os testes foram realizados, na maioria dos experimentos, mantendo as outras variáveis constantes. Os testes para variações de x foram realizados para dois casos distintos: grafos lineares (em que o grafo é uma árvore); grafos totalmente conectados.

Os testes são apresentados na próxima sessão, juntamente com sua análise.

6. Comparação da Análise de Complexidade com os Testes

Teste para variação de m

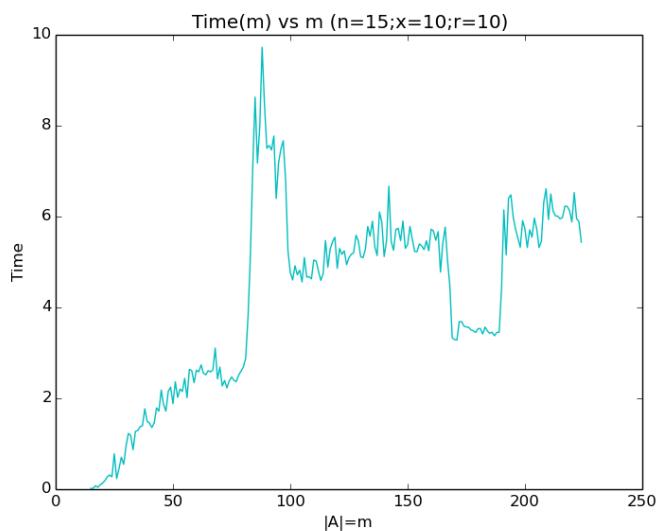


Figura 4- Tempo x m

Esse gráfico mostra que a complexidade depende de m influencia fortemente o resultado para um aumento inicial de seus valores, mas depois estabiliza. Isso se deve ao fato de, como foi analisado, os comportamentos de *LimitedDFS* tem como barreira assintótica superior $O(Q \cdot b^x)$, sendo Q uma combinação dos parâmetros constantes nesse experimento. Para um número baixo de arestas, então, esse valor b diminui e passa a ser o limitante da complexidade temporal. Para grafos com mais arestas, $\uparrow m = |A|$, esse valor é maior que os outros com os quais é comparado, na função \min , e para de influenciar o resultado, gerando um gráfico relativamente constante em m .

Teste para variação de r

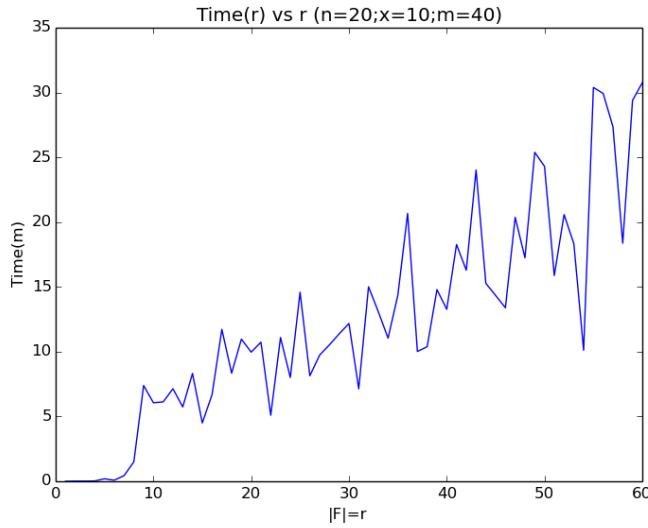


Figura 5- Tempo x r

Esse gráfico, embora com alto ruído, demonstra a relação linear de r na complexidade temporal do algoritmo.

$$O\left(r \cdot |V| \cdot \sum_{depth=1}^x \binom{|V|}{depth} + |V|^2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k}\right)$$

Para todos os parâmetros, exceto r , constantes, temos:

$$O(r)$$

Teste para variação de n em grafos esparsos

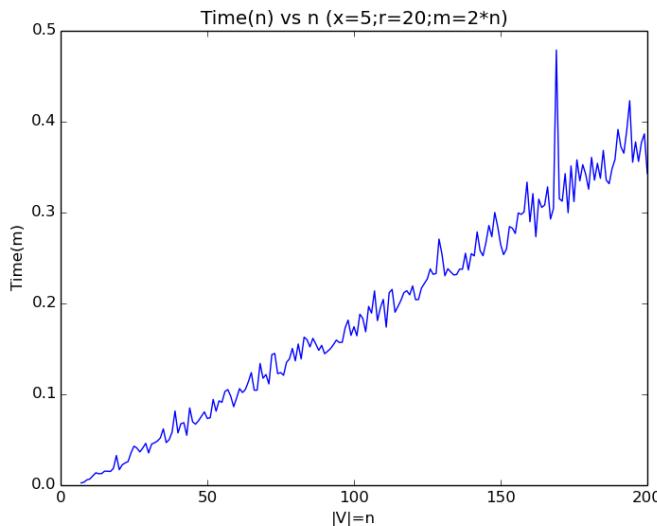


Figura 6- Tempo x n (Grafos esparsos)

Esse gráfico mostra que, para um x fixo em grafos muito esparsos, novamente:

$$b^{depth} \leq b^x$$

E, para valores consideráveis de $|V| = n$:

$$b^x \ll |V|^x \ll 2^{|V|}$$

Como esse valor b^x é pequeno e constante, a complexidade de *ExactZikaZero*, utilizando a análise completa de *LimitedDFS*, fica:

$$\begin{aligned} & O\left(\sum_{d=1}^x \min(b^d, |V| \cdot 2^{|V|-1}) + |V|^2 \cdot \min(b^d, 2^{|V|}) + \min(b^d, |V|^d) \cdot |V| \cdot r\right) \\ & O\left(\sum_{d=1}^x b^d + |V|^2 \cdot b^d + b^d \cdot |V| \cdot r\right) \\ & O\left(\sum_{d=1}^x |V|^2 + |V| \cdot r\right) \\ & O(x \cdot (|V|^2 + |V| \cdot r)) \end{aligned}$$

Como, para esse experimento, x e r são constantes:

$$comp = O(|V|^2)$$

Dada pela complexidade da função *FindAdjNodes*. Voltando, agora, à função *FindAdjNodes*, note que, como comentado na sessão 3, ela tem essa complexidade devido a dois fatores, o número de ‘filhos’ de um nó e o número de voluntários desse nó (no For das linhas 11 e 12). Como o grafo desse experimento é muito esparsos, o número de crianças de um nó será a , tal que $a \leq b$ e $a \ll |V|$. A complexidade dessa função será, então, $O(|V|)$. E a complexidade temporal do algoritmo completo passa a depender apenas linearmente de $|V| = n$, como pode ser visto no gráfico acima.

$$comp = O(|V|)$$

Teste para variação de n em grafos completamente conectados

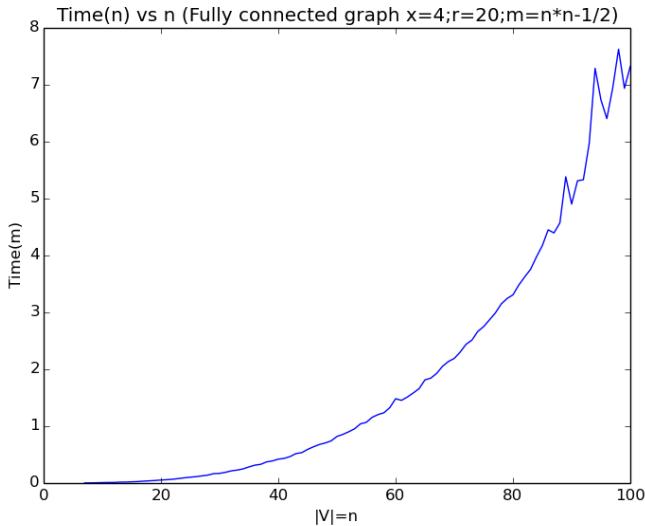


Figura 7- Tempo x n (Grafos completamente conectados)

A complexidade do algoritmo em relação a n , tendo um valor x constante, é polinomial, com grau dependendo da constante x .

$$O\left(r \cdot |V| \cdot \sum_{depth=1}^x \binom{|V|}{depth} + |V|^2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k}\right)$$

Como r é uma constante e como:

$$|V| \cdot \sum_{depth=1}^x \binom{|V|}{depth} \leq |V|^2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k}$$

Simplificamos a complexidade para:

$$O\left(|V|^2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k}\right)$$

Tendo essa equação, é fácil notar que:

$$\begin{aligned} |V|^2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k} &\leq |V|^2 \cdot \sum_{depth=1}^x depth \cdot \binom{|V|}{depth} \\ |V|^2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k} &\leq |V|^2 \cdot x^2 \cdot \binom{|V|}{x} \end{aligned}$$

E como:

$$\binom{|V|}{x} \leq |V|^x$$

$$|V|^2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k} \leq |V|^2 \cdot x^2 \cdot |V|^x = x^2 \cdot |V|^{x+2}$$

Então essa função tem complexidade temporal limitada por:

$$O(|V|^{x+2})$$

Que é polinomial, dado um x constante, assim como observado no gráfico acima.

Teste para variação de x em grafos lineares (Árvore)

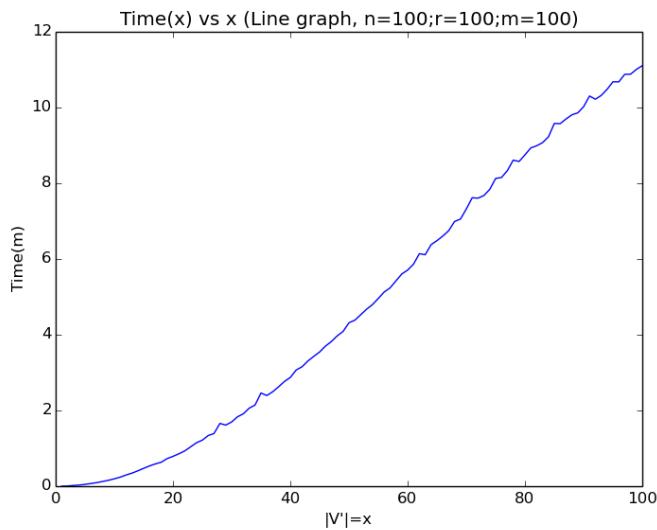


Figura 8- Tempo x X (Grafos lineares)

Novamente, para um grafo esparso, b^{depth} terá um valor baixo. Para o caso específico de um grafo linear tem-se $b = 1$, ou seja, a profundidade não afeta esse termo. A complexidade do *ExactZikaZero* para esse caso específico é, então:

$$\begin{aligned} & O\left(\sum_{d=1}^x \min(b^d, |V| \cdot 2^{|V|-1}) + |V|^2 \cdot \min(b^d, 2^{|V|}) + \min(b^d, |V|^d) \cdot |V| \cdot r\right) \\ & O\left(\sum_{d=1}^x b^d + |V|^2 \cdot b^d + b^d \cdot |V| \cdot r\right) \\ & O\left(\sum_{d=1}^x 1 + |V|^2 \cdot 1 + 1 \cdot |V| \cdot r\right) \\ & comp = O(x \cdot (|V|^2 + |V| \cdot r)) \end{aligned}$$

Obtendo-se uma complexidade linear em x , como se vê no gráfico.

Teste para variação de x em grafos completamente conectados

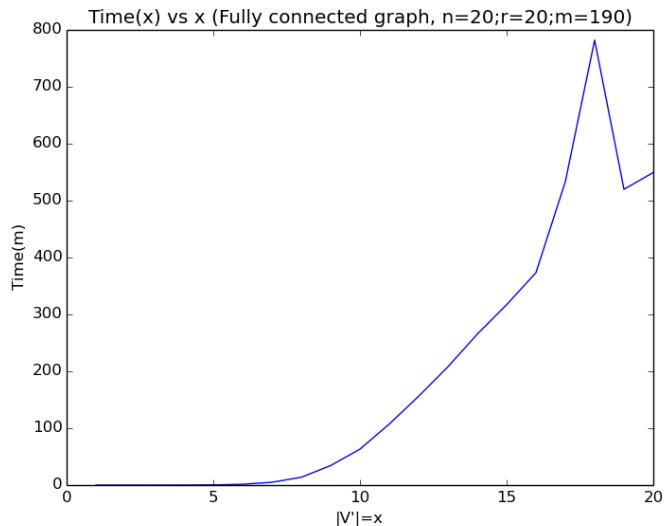


Figura 9- Tempo x X (Grafos completamente conectados)

Esse gráfico confirma a complexidade temporal encontrada anteriormente:

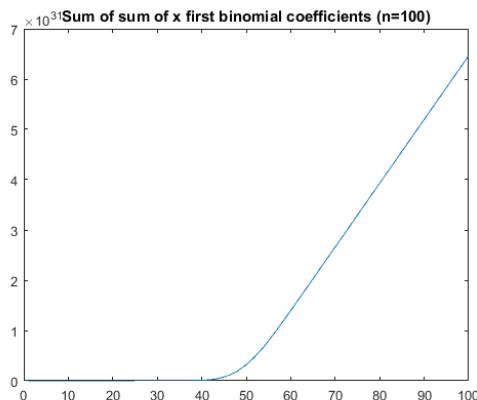
$$O\left(r \cdot |V| \cdot \sum_{depth=1}^x \binom{|V|}{depth} + |V|^2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k}\right)$$

Tendo apenas x variável:

$$O\left(c_1 \cdot \sum_{depth=1}^x \binom{|V|}{depth} + c_2 \cdot \sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k}\right)$$

$$comp = O\left(\sum_{depth=1}^x \sum_{k=0}^{depth} \binom{|V|}{k}\right)$$

Essa função está representada pelo gráfico da Figura 3, repetido aqui por praticidade, que é muito similar ao encontrado nesse experimento.



Anexos

1. Anexo 1

Pelo teorema binomial [1]:

$$(x + y)^n = \binom{n}{0} \cdot x^n \cdot y^0 + \binom{n}{1} \cdot x^{n-1} \cdot y^1 + \cdots + \binom{n}{n-1} \cdot x^1 \cdot y^{n-1} + \binom{n}{n} \cdot x^0 \cdot y^n$$

Para o caso especial em que $x = 1$ e $y = 1$:

$$(1 + 1)^n = \binom{n}{0} \cdot 1^n \cdot 1^0 + \binom{n}{1} \cdot 1^{n-1} \cdot 1^1 + \cdots + \binom{n}{n-1} \cdot 1^1 \cdot 1^{n-1} + \binom{n}{n} \cdot 1^0 \cdot 1^n$$

$$(2)^n = \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n-1} + \binom{n}{n}$$

$$2^n = \sum_{k=0}^n \binom{n}{k}$$

2. Anexo 2

Cada elemento $p_k \in P_k$ pode ter no máximo $|V| - k$ arestas, pois seriam as arestas para todo $w \in V$ tal que $w \notin p_k$ ($w \in V \setminus p_k$). Considere o subconjunto $E_k \subset E$ o conjunto de arestas que saem de algum elemento de P_k :

$$|E_k| \leq |P_k| \cdot (|V| - k)$$

Com essa inequação, pode-se calcular:

$$|E| \leq \sum_{k=0}^{|V|} |E_k|$$

$$|E| \leq \sum_{k=0}^{|V|} (|V| - k) \cdot \binom{|V|}{k}$$

$$|E| \leq \sum_{k=0}^{|V|} |V| \cdot \binom{|V|}{k} - \sum_{k=0}^{|V|} k \cdot \binom{|V|}{k}$$

$$|E| \leq |V| \cdot \sum_{k=0}^{|V|} \binom{|V|}{k} - \sum_{k=0}^{|V|} k \cdot \binom{|V|}{k}$$

Utilizando as provas dos anexos 1 e 3:

$$|E| \leq |V| \cdot 2^{|V|} - |V| \cdot 2^{|V|-1}$$

$$|E| \leq |V| \cdot 2^{|V|-1}$$

Utilizando essa equação tem-se um limite assintótico para $|E|$:

$$|E| \in O(|V| \cdot 2^{|V|})$$

3. Anexo 3

Pelo teorema binomial [1], para $y = 1$:

$$(x + 1)^n = \binom{n}{0} \cdot x^0 + \binom{n}{1} \cdot x^1 + \cdots + \binom{n}{n-1} \cdot x^{n-1} + \binom{n}{n} \cdot x^n$$
$$(x + 1)^n = \sum_{k=0}^n \binom{n}{k} \cdot x^k$$

Derivando ambos os lados em x :

$$n \cdot (x + 1)^{n-1} = \sum_{k=1}^n k \cdot \binom{n}{k} \cdot x^{k-1}$$

Substituindo $x = 1$:

$$\sum_{k=0}^n k \cdot \binom{n}{k} = n \cdot 2^{n-1}$$

4. Anexo 4

Prova retirada de [2].

$$\sum_{j=1}^n \sum_{k=0}^j \binom{n}{k} = \sum_{j=0}^n \sum_{k=0}^j \binom{n}{k} - \binom{n}{0}$$
$$\sum_{j=1}^n \sum_{k=0}^j \binom{n}{k} = \sum_{k=0}^n \sum_{j=k}^n \binom{n}{k} - 1$$
$$\sum_{j=1}^n \sum_{k=0}^j \binom{n}{k} = \sum_{k=0}^n \binom{n}{k} \cdot \sum_{j=k}^n 1 - 1$$
$$\sum_{j=1}^n \sum_{k=0}^j \binom{n}{k} = \sum_{k=0}^n \binom{n}{k} \cdot (n - k + 1) - 1$$
$$\sum_{j=1}^n \sum_{k=0}^j \binom{n}{k} = (n + 1) \cdot \sum_{k=0}^n \binom{n}{k} - \sum_{k=0}^n k \cdot \binom{n}{k} - 1$$

Utilizado os Anexos 1 e 3:

$$\sum_{j=1}^n \sum_{k=0}^j \binom{n}{k} = (n + 1) \cdot 2^n - n \cdot 2^{n-1} - 1$$
$$\sum_{j=1}^n \sum_{k=0}^j \binom{n}{k} = (n + 2) \cdot 2^{n-1} - 1$$

Referências

- [1] E. W. Weisstein, “Binomial Theorem.,” MathWorld--A Wolfram Web Resource, [Online]. Available: <http://mathworld.wolfram.com/BinomialTheorem.html>. [Acesso em 29 04 2016].
- [2] B. M. Scott, “Sum of sum of binomial coefficients,” Mathematics Stack Exchange, [Online]. Available: <http://math.stackexchange.com/q/1766037> . [Acesso em 30 04 2016].

UFMG/ICEx/DCC
Pós-Graduação em Ciência da Computação
Projeto e Análise de Algoritmos

Trabalho Prático I

Jessica Sena de Souza

Problema

Dados um grafo $G(\mathbb{V}, \mathbb{A})$, em que \mathbb{V} é o conjunto de n voluntários e \mathbb{A} é o conjunto de m laços de amizade, um conjunto \mathbb{F} dos r focos de reprodução do mosquito, e, uma relação $R(v) : \mathbb{V} \rightarrow \mathbb{F}$, definida para cada $v \in \mathbb{V}$, explicitando os focos de reprodução aos quais o voluntário v tem acesso. O objetivo é selecionar o menor número de voluntários $\mathbb{V}' \subset \mathbb{V}$, tal que, todo foco é acessado, por pelo menos um voluntário $v \in \mathbb{V}'$ e o grafo induzido por \mathbb{V}' em G é conexo.

Exercício 1

A modelagem do problema seguiu a descrição de tal forma que no grafo $G(\mathbb{V}, \mathbb{A})$, \mathbb{V} permanece sendo o conjunto de n voluntários e \mathbb{A} permanece sendo o conjunto de m laços de amizade. Ou seja, os vértices representam os voluntários e as arestas os laços de amizade entre esses voluntários. A relação $R(v) : \mathbb{V} \rightarrow \mathbb{F}$ foi modelada em um vetor de vértices onde cada posição representa os focos aos quais o voluntário v tem acesso.

Exercício 2

Para solucionar o problema ZikaZeroZ, inicialmente é necessário gerar todos os sub conjuntos \mathbb{V}' para dar a solução ótima. O Algoritmo 1 apresenta solução para gerar todos os subconjuntos.

Algorithm 1 Gerador de Subconjuntos Gera todos os subconjuntos \mathbb{V}' para um grafo $G(\mathbb{V}, \mathbb{A})$

Require: Inicio i

```
1: function GERADORSUBCONJUNTOS( $i$ )
2:   if  $i > \mathbb{V}$  then
3:     Adiciona  $\mathbb{V}'$  em Subconjuntos
4:     return Subconjuntos
5:   else
6:     Insere  $i$  em  $\mathbb{V}'$ 
7:     GeradorSubconjuntos( $i + 1$ )
8:     Retira  $i$  de  $\mathbb{V}'$ 
9:     GeradorSubconjuntos( $i + 1$ )
```

Após a geração dos subconjuntos é usada uma heurística que ordena os subconjuntos por tamanho, afim de testar primeiro os subconjuntos menores dado que a solução do problema envolve achar o menor subconjunto possível. Assim o primeiro subconjunto que for encontrado que respeite as regras de ser conexo e de cobrir todos os vértices deverá ser obrigatoriamente a solução ótima. O Algoritmo 2 demonstra a casca da solução para verificar se o grafo \mathbb{V}' é conexo. Esse algoritmo chama uma busca em largura(Ver Algoritmo 3) para verificar se o sub grafo formado pelo subconjunto de vértices é conexo. O Algoritmo 4 demonstra a solução usada para verificar se o subconjunto de vértices \mathbb{V}' cobre todos os r focos.

Algorithm 2 Conexo Verifica se o subconjunto \mathbb{V}' é conexo.**Require:** Subconjunto \mathbb{V}'

```
1: function CONEXO( $i$ )
2:    $G(\mathbb{V}', \mathbb{A}') \leftarrow \mathbb{V}'$                                  $\triangleright$  Cria um grafo representando o subconjunto de vértices
3:   BFS( $G(\mathbb{V}', \mathbb{A}')$ )
4:   if Todos  $\mathbb{V}'$  vértices de  $G$  foram alcançados pela BFS then
5:     return True
6:   else
7:     return False
```

Algorithm 3 BFS Percorre o grafo $G(\mathbb{V}, \mathbb{A})$ marcando os vértices visitados**Require:** Grafo $G(\mathbb{V}, \mathbb{A})$

```
1: function BFS( $i$ )
2:    $u \leftarrow 0$                                           $\triangleright$  Assinala como vértice inicial o vértice 0
3:    $V[u] = Visitado$                                       $\triangleright$  Marca o vértice inicial como visitado
4:   Insere  $u$  na Fila  $F$ 
5:   while  $F$  tem elementos do
6:      $u \leftarrow$  primeiro elemento de  $F$ 
7:     for all Vértice  $v \in G.Adj[u]$  do
8:       if  $v$  não foi visitado then
9:          $V[v] = Visitado$ 
10:        Insere  $v$  na Fila  $F$ 
11:        Retira  $u$  na Fila  $F$ 
12:   return  $V$ 
```

Algorithm 4 Cobre Focos Verifica se o subconjunto \mathbb{V} cobre os r focos**Require:** Subconjunto de vértices \mathbb{V}

```
1: function COBREFOCOS( $i$ )
2:   for all Vértice  $v \in \mathbb{V}$  do
3:     for all Foco  $f$  coberto por  $v$  do
4:       if  $f$  não foi coberto then
5:          $Focos[v] = True$                                 $\triangleright$  Marca o foco  $f$  como coberto
6:   if  $r$  focos foram cobertos then
7:     return True
8:   else
9:     return False
```

A respeito do critério de desempate para quando houver mais de uma solução ótima, foi feito uma escolha baseada na primeira solução ótima encontrada, uma vez que todos os subconjuntos são gerados e ordenados crescentemente, a solução que for encontrada e respeitar os critérios de cobrir todos os focos e ser conexo o sub conjunto, será obrigatoriamente a que contém menos voluntários.

Exercício 3

A discussão de complexidade Temporal e Espacial será feita para cada algoritmo separadamente nas sub seções a seguir.

Complexidade Temporal

O tempo de execução do algoritmo **Gerador de Subconjuntos**(1) é dominado pelas chamadas recursivas das linhas 7 e 9. No **melhor caso** sua complexidade é $O(1)$ quando o grafo $G(\mathbb{V}, \mathbb{A})$ não possui vértices e portanto só existe um subconjunto \mathbb{V}' que é o subconjunto vazio. No **pior caso** são gerados todos os subconjuntos de $G(\mathbb{V}, \mathbb{A})$ tendo complexidade de $O(2^{|\mathbb{V}|})$.

O tempo de execução do algoritmo **Conexo**(2) é dominado pela linha 3 onde é feito a chamada do algoritmo **BFS**(3). BFS, por sua vez, é dominado assintoticamente pelas linhas 5 a 11 onde todos os vértices e arestas do grafo são percorridos. Seu tempo de execução é $O(V + A)$, onde V é o número de vértices e A é o número de arestas. Essa complexidade é dominada pelo termo A , que no **pior caso** degenera para V^2 .

Por fim, o tempo de execução do algoritmo **Cobre Focos**(4) é dominado assintoticamente pelas linhas 2 a 5. Sua complexidade de tempo é de $O(|\mathbb{V}'| \cdot r)$ onde r é o numero de focos.

Complexidade Espacial

No **melhor caso** do algoritmo **Gerador de Subconjuntos** acontece quando não existe vértices em $G(\mathbb{V}, \mathbb{A})$ e somente um subconjunto é armazenado e sua complexidade de espaço é $O(1)$. No **pior caso**, todos os subconjuntos são armazenados levando a uma complexidade de espaço de $O(2^{|\mathbb{V}|})$.

O algoritmo **Conexo** tem complexidade de espaço de $O(|\mathbb{V}'|^2)$ uma vez que cria o subgrafo $G(\mathbb{V}', \mathbb{A}')$ usando matriz de adjacência. O algoritmo **BFS** tem sua complexidade de espaço de $O(2^{|\mathbb{V}'|})$, dado que no **pior caso** armazena o número de vértices duas vezes, uma para marcar quais foram visitados e outra vez na fila de escolha de vértices para serem percorridos.

O algoritmo **Cobre Focos** tem complexidade de espaço de $O(r)$, onde r é o número de focos possíveis de serem atingidos.

Exercício 4

O algoritmo proposto foi implementado na linguagem C++ e se encontra no zip juntamente com essa documentação. O algoritmo está modularizado em duas classes (Graph e ZikaZeroZ) e um arquivo principal (main).

Exercício 5

Os resultados são médias de 5 execuções e foram executados em um Pentium dual-core com 4GB de RAM. As soluções foram analisadas variando o número de voluntários (número de vértices do grafo $G(\mathbb{V}, \mathbb{A})$) e o número de focos de Zika Virus. O número de linhas foi mantido fixo em 10. Na figura 1 são mostrados os gráficos do tempo de execução com o aumento do número de voluntários no problema. Foi acrescido a esse experimento a fixação de dois valores distintos de número de focos para verificar o comportamento da solução. A solução não mostrou variação em tempo de execução quando o número de focos dobrou.



Figura 1: Gráfico de variação do número de voluntários (vértices) versus o tempo de execução (em segundos). No gráfico da esquerda, o número de focos de Zika foi fixado em 5 e no gráfico da direita, foi fixado em 10 focos.

Exercício 6

A curva, mostra que o comportamento do tempo de execução está de acordo com a análise de complexidade do algoritmo **Subconjunto(1)** que domina assintoticamente a complexidade de toda a solução. O experimento do exercício 5 foi feito variando o número de voluntários, por este ser o fator que mais impacta no tempo de execução do algoritmo proposto. Porém, além disso, foi feito um experimento variando o número de focos afim de comparar a análise assintótica feito a respeito do algoritmo **Cobre Focos(4)**. A Figura 2 mostra crescimento linear da curva, o que está de acordo com a complexidade analisada $O(|V'| \cdot r)$, uma vez que $|V'|$ permaneceu fixo, e houve variação linear de r .

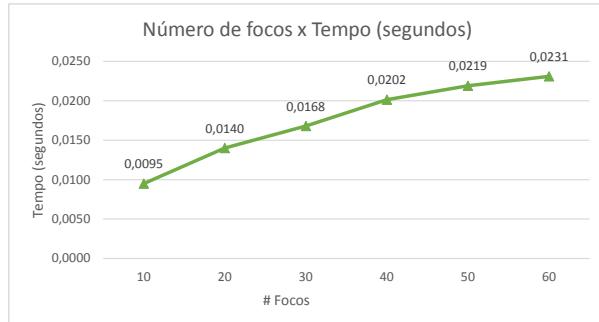


Figura 2: Gráfico de tempo de execução do algoritmo ao fixar o número de vértices e variar o número de focos.

Referências

- [1] Robert Sedgewick “Algorithms in C++ Part 5: Graph Algorithms”. Addison-Wesley Professional, (3rd Edition), 2006.
- [2] Steven S. Skiena “The Algorithm Design Manual”. Springer, 2nd edition (July 26, 2008).

Programa de Pós-Graduação em Ciência da Computação - DCC/UFMG
Projeto e Análise de Algoritmos

Trabalho Prático: ZikaZeroZ

Nícollas de Campos Silva

Universidade Federal de Minas Gerais (UFMG)
ncsilvaa@gmail.com

1 de maio de 2016

Resumo

Este documento tem como objetivo descrever a modelagem e solução proposta para o problema ZikaZeroZ, bem como justificar as principais decisões e escolhas associadas a tal solução. Além disso, discutimos o impacto de algumas instâncias do problema a fim de analisar a eficiência do algoritmo proposto.

1 Modelagem Proposta

A modelagem proposta consiste em um grafo $G = (V, E)$ não orientado, onde os voluntários na ação contra o *zika-vírus* representam o conjunto V de vértices e os laços de amizades existentes entre eles o conjunto E de arestas do grafo. Para tal abordagem, optou-se por utilizar o conceito de matriz de adjacências na representação do grafo, visto que o principal objetivo deste trabalho consiste apenas em encontrar uma solução ótima, sem restrições a eficiência. Baseado no conceito apresentado em Cormen (2002), a representação de uma matriz de adjacência de um grafo G consiste em uma matriz $A_{|V| \times |V|} = a_{ij}$, tal que:

$$a_{ij} = \begin{cases} 1, & \text{se } (i, j) \in E, \\ 0, & \text{caso contrário.} \end{cases}$$

De maneira análoga, o mesmo conceito de uma matriz de adjacências foi utilizado para representar se um voluntário tem ou não acesso a um determinado foco do *zika-vírus*. Construímos uma matriz de focos $B_{|V| \times |F|} = b_{ij}$, tal que:

$$b_{ij} = \begin{cases} 1, & \text{se o voluntário tem acesso ao foco,} \\ 0, & \text{caso contrário.} \end{cases}$$

A escolha da matriz de adjacências como a abordagem a ser utilizada se deve a praticidade e simplicidade de utilização, principalmente quando os grafos são razoavelmente pequenos. Além disso, como o grafo é não ponderado, existe uma vantagem adicional em termos de espaço de armazenamento, uma vez que utilizamos apenas um bit por entrada de dados.

A figura 1 abaixo demonstra a utilização de nossa modelagem para o grafo proposto como instância inicial para o problema *zikazeroZ*. Podemos notar que do grafo original resulta a matriz de adjacências (b) e a matriz relativa aos focos da instância (c).

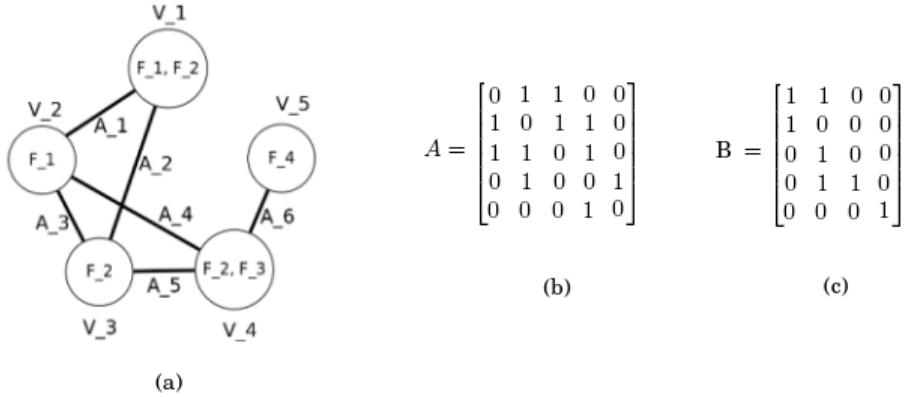


Figura 1: Aplicação da modelagem sobre a instância proposta na especificação deste trabalho. Em (a) temos o grafo proposto. (b) mostra a matriz de adjacências resultante. (c) mostra a matriz relacionando os voluntários aos focos.

2 Solução Proposta

O objetivo deste trabalho é selecionar o menor número de voluntários $V' \subset V$, tal que, todos os focos são acessados por pelo menos um voluntário $v \in V'$, de forma que o grafo induzido por V' em G é conexo. Em outras palavras, temos sempre que encontrar a solução ótima para o problema, ou seja, a solução com o menor número de voluntários dentre todas as combinações possíveis. Dessa forma, o algoritmo proposto consiste em analisar todas as possíveis combinações de vértices, construindo sempre um novo grafo $G' \subset G$, partindo do pressuposto que apenas as componentes conexas de G poderão ser classificadas como possíveis soluções. Neste intuito, dividimos a tarefa de estimar uma solução viável para o problema em três subtarefas menores:

1. Estimar se o grafo G' construído é uma componente conexa de G .
2. Estimar se os vértices (voluntários) de G' atingem todos os r focos de *zika-vírus*.
3. Analisar quantos vértices (voluntários) existem nessa solução encontrada.

O algoritmo proposto está implementado da maneira com que se segue. Dado um grafo $G = (V, E)$, o algoritmo computa todos os subgrafos possíveis, gerados a partir de G , incluindo a cada iteração um novo vértice no grafo G' . Basicamente, analisamos todas as $2^{|V|}$ combinações existentes no grafo G , sempre construindo um novo grafo G' . Em seguida, são realizadas todas as subtarefas apresentadas anteriormente. Na primeira subtarefa, analisamos se o grafo G' construído é uma componente conexa de G por meio da aplicação da estratégia de busca em profundidade. Especificamente, aplicamos a busca em profundidade no primeiro vértice s existente em G' a fim de verificar se todos os vértices $v' \in V'$ são alcançáveis por s . Caso todos os vértices sejam alcançados nessa busca, dizemos que G' é uma componente conexa de G . Caso contrário, descartamos o grafo G' e procuramos por uma nova solução. Com essa abordagem inicial, conseguimos reduzir o número de soluções possíveis de $2^{|V|}$ combinações para apenas as combinações que são componentes conexas de G .

A segunda subtarefa consiste em analisar todas as soluções resultantes após a filtragem realizada pela primeira subtarefa. De forma simples, analisamos se todos os vértices $v'_i \in V'$ são capazes de cobrir todos os focos de *zika-vírus* existentes. Para tal, analisamos a matriz de focos construída verificando as linhas correspondentes aos vértices pertencentes a solução G' válida até este momento. Após a aplicação dessa subtarefa, reduzimos o número de soluções possíveis de $2^{|V|}$ combinações para apenas as combinações que são componentes conexas de G e que cobrem todos os focos.

Por fim, a terceira subtarefa consiste em analisar todas as soluções resultantes após as filtragens realizadas pelas subtarefas 1 e 2. Basicamente, para cada solução possível existente procuramos por aquela que possua o menor número de vértices, de forma a minimizar o número de voluntários necessários na ação contra o *zika-vírus*. Após essa subtarefa teremos uma ou mais soluções com o menor número de vértices possível, de forma que a solução G' seja um grafo conexo que cobre todos os focos. O critério de desempate no caso de soluções com o mesmo número de vértices consiste em encontrar a solução cuja soma dos índices dos vértices é a menor possível.

O algoritmo 1 especifica os detalhes da implementação, deixando claro as três chamadas de funções feitas para realizar as subtarefas. É importante destacar que as combinações foram geradas baseado em um contador binário que permuta o conjunto C por $2^{|V|}$ vezes, sempre encontrando subgrafos de G .

Algorithm 1 ZikaZeroZ Problem

```

1: best  $\leftarrow \infty$ 
2: solution  $\leftarrow \infty$ 
3: for each combination  $C \in 2^{|V|}$  do
4:    $G' \leftarrow C$ 
5:   if connected( $G'$ ) then
6:     if focusCoverage( $G'$ ) then
7:       solution = volunteers( $G'$ )
8:       if solution < best then
9:         best  $\leftarrow$  solution
10:        bestSolution  $\leftarrow G'$ 
11:      end if
12:    end if
13:  end if
14: end for

```

Por sua vez, os algoritmos 2 e 3 representam a forma com que a primeira subtarefa foi implementada. Em 2, notamos que a estratégia da busca em profundidade foi implementada com base em um procedimento recursivo que marca os vértices visitados, desde que exista um caminho de um vértice a outro, conforme verifica a linha 6. Em 3, fica claro a forma com que podemos utilizar busca em profundidade para verificar se G' é um grafo conexo, e assim uma componente conexa de G . A partir de um vértice s qualquer de G' (linha 1), se, ao aplicarmos busca em profundidade em s (linha 2), notarmos que algum outro vértice v_i de G' (linhas 3-7) não foi visitado por s , temos um grafo desconexo e assim inválido para nossa solução. Caso contrário, G' pode ser uma possível solução.

Algorithm 2 DFS(vertex s)

```

1: if visited[s] then
2:   return
3: end if
4: visited[s]  $\leftarrow 1$ 
5: for each  $v_i \in V'$  do
6:   if  $s \rightsquigarrow v_i$  then DFS( $v_i$ )
7:   end if
8: end for

```

A figura 2 mostra como seria cada procedimento quando aplicado para um grafo qualquer, como mostrado em (a), que possui 3 voluntários (vértices), 2 ligações de amizade entre eles (arestas) e 2 focos de *zika-vírus*. Primeiramente, todas as combinações possíveis são enumeradas

Algorithm 3 CONNECTED()

```

1:  $s \leftarrow \text{vertex } G'$ 
2: DFS(vertex s)
3: for each  $v_i \in V'$  do
4:   if  $v_i \in G'$  then
5:     if  $v_i \notin \text{visited}$  then return FALSE
6:   end if
7: end if
8: end for
9: return TRUE

```

pela estratégia do contador binário, como mostrado em (b). Em seguida, cada subtarefa realiza um processo de refinamento nos resultados, como mostrado em (c), até que poucas possíveis soluções são avaliadas no processo final. Ao final do processo, temos um grafo G' conexo que satisfaz as condições do problema.

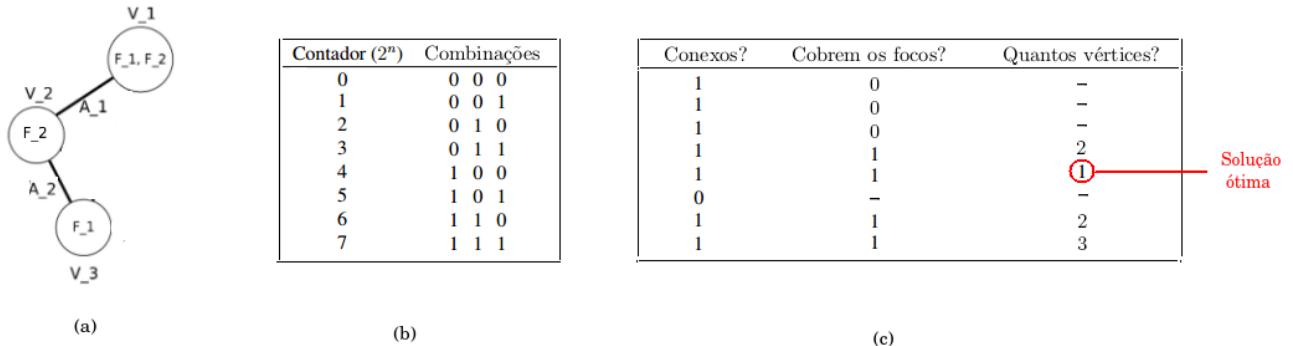


Figura 2: Aplicação da solução proposta sobre um exemplo aleatório: (a) Grafo proposto; (b) combinações possíveis a serem analisadas pelo algoritmo; (c) resultado da aplicação de cada uma das subtarefas relacionadas.

3 Análise da Complexidade Temporal e Espacial

No intuito de analisarmos previamente o desempenho do algoritmo implementado, avaliamos o custo computacional de cada procedimento, bem como o custo de armazenamento gerado por cada rotina executada. Dessa forma, devemos analisar cada trecho do algoritmo 1 proposto, nos retendo especificamente as subtarefas propostas e analisando as funções que dominam assintoticamente as demais.

O primeiro ponto a ser observado é que cada subtarefa no algoritmo 1 executa $2^{|V|}$ combinações possíveis (linha 3), uma vez que analisamos cada subgrafo G' de G . O custo de gerar um subgrafo G' (linha 4), de acordo a combinação C , é o custo de copiarmos a matriz de adjacências para cada C distinto. Tal operação tem custo $O(V^2)/2$ uma vez que são copiadas apenas as posições referentes a matriz triangular superior de G , visto que a matriz de adjacências é simétrica. Por sua vez, o custo de armazenamento é $O(|V|^2)$, pois todo grafo G' alocado é também desalocado no final de cada iteração.

A primeira subtarefa, de verificar se o grafo G' é conexo (linha 5), consiste na realização do algoritmo 3. O custo computacional desse algoritmo consiste na realização da busca em profundidade para um vértice qualquer de G' (linha 2 do algoritmo 3) e na verificação de todos os vértices de G' (linha 3 do algoritmo 3). O custo da busca em profundidade é $O(|V'| + |M'|)$,

onde V' e M' são, respectivamente, os vértices e as arestas de G' . Por sua vez, o custo de verificar cada vértice V' consiste em $O(V')$. Assim, o algoritmo 3 tem um custo assintótico de $O(|V'| + |V'| + |M'|) = O(|V'| + |M'|)$. O custo de armazenamento consiste apenas em $O(V)$ visto que mantemos apenas o vetor *visited*.

A segunda subtarefa, que consiste em analisar se o grafo G' cobre todos os r focos, tem custo computacional de $O(|V| * |R|)$. Tal custo deve-se ao fato de que para realizar a subtarefa devemos checar todas as $|V| * |R|$ posições da matriz de adjacências que representa os focos. Por sua vez, o custo de armazenamento consiste $O(|R|)$, uma vez que apenas armazenamos um vetor para marcar quais focos foram atingidos ou não. Como última análise, a terceira subtarefa, de retornar o número de vértices existentes em G' , tem custo computacional $O(|V|)$, visto que temos apenas que varrer o vetor relativo a combinação que originou G' . Esta tarefa não possui custo de armazenamento.

Dessa forma, a complexidade temporal de todo o algoritmo 1 consiste em considerarmos todas as complexidades calculadas aqui, considerando as $2^{|V|}$ combinações possíveis em que cada subtarefa é executada. Em contrapartida, a complexidade espacial não depende das $2^{|V|}$ em que o algoritmo é executado, uma vez que as matrizes e vetores são alocados e desalocados da memória a cada iteração. Por isso, o custo espacial consiste na matriz de adjacências, a matriz relativa aos focos, e todos os custos citados anteriormente, relativos a cada subtarefa.

- **Temporal:** $2^{|V|} * O(V^2)/2 + O(|V'| + |M'|) + O(|V| * |R|) + O(|V|) = O(2^{|V|} * (|V|^2 + |V||R|))$
- **Espacial:** $O(|V|^2) + O(|V| * |R|) + O(|V|) + O(|R|) = O(|V|^2 + |V||R|)$.

4 Implementação do Algoritmo

O algoritmo proposto na seção 2 foi implementado na linguagem C++, devido a simplicidade, praticidade e eficiência da linguagem. Não foi utilizado nenhum recurso computacional prático, como bibliotecas que contém procedimentos já implementados para a manipulação de grafos. Para facilitar o processo de compilação e execução desse algoritmo, foram criados os scripts na linguagem shell. Todos os códigos e exemplos utilizados seguem em anexo, junto deste arquivo.

5 Estudo de Caso

No intuito de avaliar a eficiência, corretude e desempenho do algoritmo proposto, analisamos a aplicação do modelo proposto para diversos cenários. Especificamente, consideramos exemplos de casos elaborados, bem como casos consideravelmente grandes. Pretendemos também, avaliar para quais instâncias o algoritmo leva um tempo considerável para encontrar uma solução ótima. De maneira geral, os testes foram executados apenas uma vez, pelo fato do algoritmo 1 ser determinístico, em um computador com 2GB de memória RAM e um processador Pentium(R) Dual-Core com 2.30GHz.

Na primeira análise, executamos o algoritmo para os exemplos mostrados na figura 3, avaliando a corretude deste para três diferentes modelos de grafo. Para esta análise, consideramos a mesma quantidade de focos para todos os modelos e também não se importando com a quantidade de vértices ou arestas. Variamos apenas a forma com que o grafo está organizado. Consideramos um grafo formato estrela, conforme mostra em 3(a), um grafo bipartido 3(b) e um grafo desconexo 3(c).

Após a aplicação do algoritmo notamos que, mesmo com diferentes modelos, os resultados correspondem ao esperado. Mesmo a aplicação do grafo desconexo 3(c), não influencia no algoritmo, uma vez que este ainda é capaz de encontrar uma componente conexa neste grafo. Os outros dois formatos, também não influenciaram nos resultados esperados. A figura 4 evidencia os resultados ótimos fornecidos pelo algoritmo, por meio dos vértices pintados na cor cinza.

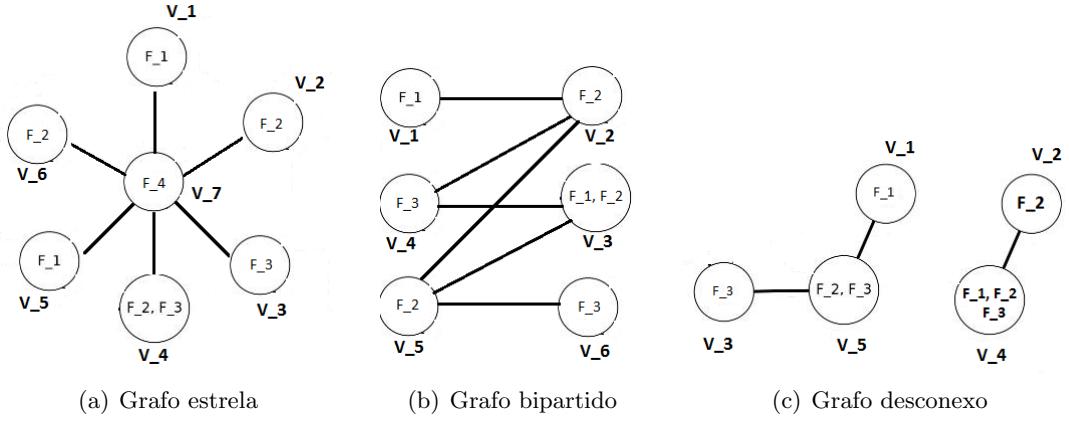


Figura 3: Variação dos modelos dos grafos utilizados.

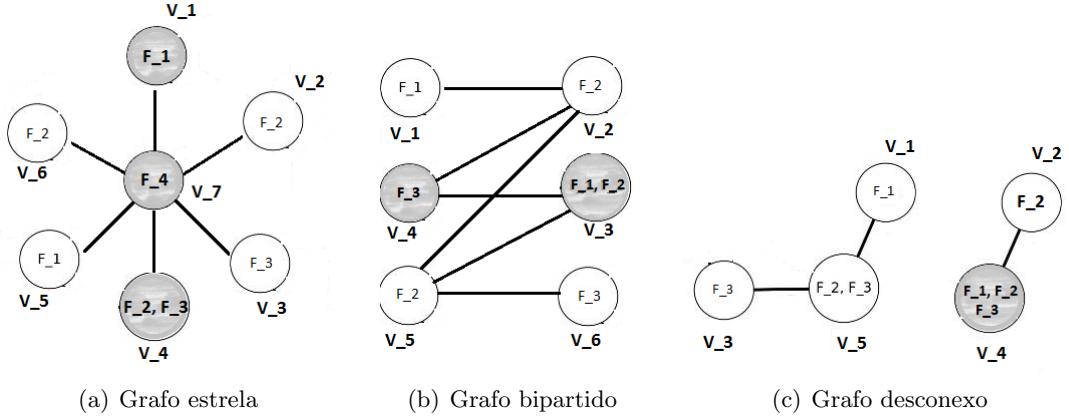


Figura 4: Resultados da aplicação do algoritmo nos modelos.

Por outro lado, avaliamos o desempenho do algoritmo para cenários com maior número de vértices e focos. Para tal, variamos o número de vértices e plotamos o gráfico 5(a), relacionando o tamanho dos vértices com o tempo de execução do algoritmo. De maneira análoga, variamos o número de focos 5(b) e, novamente, avaliamos a relação do número de focos com o tempo de execução. Para essa abordagem, medimos o tempo de execução por meio da função *time* do *linux*.

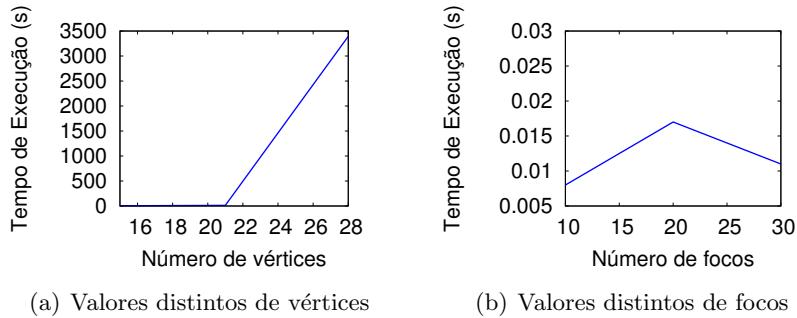


Figura 5: Resultados da aplicação do algoritmo variando o tamanho da entrada.

Conforme podemos notar, ao variar o número de vértices do grafo, que o tempo de execução do algoritmo capaz de resolver o problema *ZikaZeroZ* cresce exponencialmente com relação a entrada dos dados. A cada vez que aumentamos o número de vértices ou focos do problema, o algoritmo se torna impraticável. Ao se utilizar 28 vértices em uma determinada instância,

é necessário um processamento de aproximadamente 1 hora. Por outro lado, ao analisarmos a relação do tempo de execução com o número de focos do grafo, vemos que este não influencia muito nas decisões. Tal análise nos permite comprovar que os vértices são os fatores mais relevantes no desempenho computacional do algoritmo.

6 Análise dos Resultados

Com base nos resultados observados podemos notar que o problema *ZikaZeroZ* é um problema NP-Difícil, uma vez que para algumas instâncias ele se torna impraticável. As $2^{|V|}$ combinações necessárias para obtermos uma resposta ótima, faz com que o algoritmo seja classificado na classe dos algoritmos exponenciais. Tal observação se torna visivelmente clara quando comparamos o desempenho do algoritmo com a análise de complexidade feita na seção 3. Ao afirmarmos que o algoritmo é $O(2^{|V|} * (|V|^2 + |V||R|))$, podemos dizer que existe uma constante positiva c_1 que faz com que:

$$2^{|V|} * (|V|^2 + |V||R|) \leq c_1 * (2^{|V|} * (|V|^2)) \quad (1)$$

Tal determinação, propõe um limite assintótico válido para uma constante $c_1 \geq \frac{|V||R|}{|V|}$, para V e R como valores inteiros e positivos. Dessa forma, podemos notar que de fato as análises de complexidade resultam nos tempos de execução dos algoritmos propostos.

De maneira geral, podemos dizer que o algoritmo possui um tempo inviável para ser executado em um cenário real devido ao alto custo computacional. Com isso, a aplicação de heurísticas de podas são os passos necessários almejados para os trabalhos futuros.

Referências

Cormen, T. H. (2002). *Algoritmos: teoria e prática*. Elsevier.

Trabalho Prático 1

Projeto e Análise de Algoritmos

Bárbara G. C. O. Lopes¹

¹Universidade Federal de Minas Gerais (UFMG)
Instituto de Ciências Exatas – Departamento de Ciência da Computação
Belo Horizonte – Minas Gerais – Brasil

Resumo. Este trabalho apresenta o relatório do trabalho prático do módulo de grafos da disciplina de Projeto e Análise de Algoritmos (PAA), da Universidade Federal de Minas Gerais (UFMG), que propõe o problema ZikaZeroZ. Esse problema foi resolvido com a utilização de grafos. Será apresentada a solução desenvolvida para o problema, assim como suas características, complexidades e resultados.

1. Exercício 1 - Modelagem

O problema do ZikaZeroZ foi modelado como um grafo $G(\mathbb{V}, \mathbb{A})$, no qual \mathbb{V} é o conjunto de n voluntários e \mathbb{A} é o conjunto de m laços de amizade entre eles. É definido um conjunto \mathbb{F} dos r focos de reprodução do mosquito, e uma relação $R(v) : \mathbb{V} \rightarrow \mathbb{F}$ para cada $v \in \mathbb{V}$, explicitando os focos de reprodução aos quais o voluntário v tem acesso. A Figura 1 ilustra um exemplo de instância para o problema.

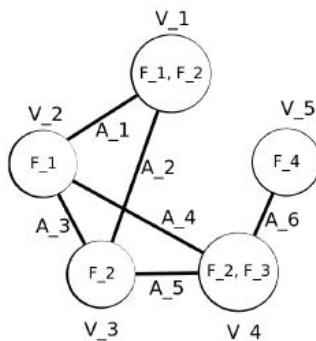


Figura 1. Exemplo de instância do problema ZikaZeroZ

O objetivo é selecionar o menor número de voluntários $\mathbb{V}' \subset \mathbb{V}$, de forma que todo foco seja acessado por, pelo menos, um voluntário $v \in \mathbb{V}'$ e que o grafo induzido por \mathbb{V}' em G seja conexo. Um exemplo de solução é ilustrado na Figura 2.

O grafo é lido através de um arquivo de entrada, a ser recebido como parâmetro, de forma a popular um mapa de adjacências (Figura 3), que para cada vértice $v \in \mathbb{V}$, armazena uma lista de vértices adjacentes a v . Também será populada uma lista de focos (Figura 4), que para cada posição, referente a um vértice $v \in \mathbb{V}$, armazenará uma lista contendo os focos ao qual o vértice tem acesso. O formato do arquivo de entrada deve respeitar às restrições indicadas na Figura 3 e os números contidos em cada linha dele devem ser inteiros.

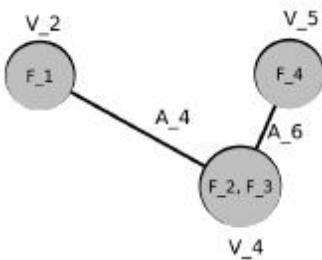


Figura 2. Solução da instância do problema exemplo

```

n m
<m-linhas-relativas-aos-laços-de-amizade>
  <cada linha contém dois índices dos voluntários, separados por um espaço>
r
<n-linhas-relativas-aos-focos-que-cada-voluntário-tem-acesso>
  <cada linha contém os índices dos focos
  que o voluntário tem acesso, separados por um espaço>

```

Figura 3. Formato do arquivo de entrada

Para a implementação da solução, considerou-se que o grafo descrito no arquivo de entrada será não direcionado e não ponderado. Portanto, as arestas serão espelhadas ($Aresta(u,v) = Aresta(v,u)$).

Optou-se pela utilização de uma estrutura de mapa para representar as adjacências, pois com ela será possível encontrar determinado nodo em $O(1)$, e, devido a utilização da linguagem Java, na qual a primeira posição de um vetor é 0 e não 1. O mapa de adjacências armazena, para cada $v \in \mathbb{V}$, uma key (o número do vértice) e um value (a lista de adjacências deste vértice).

O conjunto de focos acessados por cada vértice será armazenado em uma estrutura de lista, onde cada posição armazenará um conjunto de focos para determinado vértice.

Serão geradas combinações simples dos vértices do grafo em ordem crescente, variando p de 1 a n , sendo p a quantidade de elementos da combinação a ser gerada e n a quantidade de vértices de G , de forma a se obter subgrafos do mesmo.

O primeiro subgrafo encontrado que contenha todos os focos e seja conexo será retornado como solução para o problema e salvo no arquivo de saída (Figura 4), pois, como as combinações se iniciam com o menor tamanho (1) até o maior (n), garante-se que não haverá outra solução com um número menor de vértices do que a já encontrada.

```

<única-linha>
  <contendo os voluntários selecionados como solução ao Problema ZikaZeroZ,
  em-ordem-crescente, separados por um espaço>

```

Figura 4. Formato do arquivo de saída

Para verificar se a combinação atende às precondições, primeiramente é verificado se ela possui acesso à todos os focos. Caso os vértices desta combinação tenham acesso a todos os focos, é verificado se este subgrafo é conexo, ou seja, se para cada $v \in \mathbb{V}$ há

um caminho para qualquer outro $v \in \mathbb{V}$. Isso será verificado através da execução de uma Busca em Largura no subgrafo gerado.

1.1. Combinação Simples

Na combinação simples, a ordem dos elementos no agrupamento não interfere ($1, 2 = 2, 1$). São arranjos que se diferenciam somente pela natureza de seus elementos. A combinação simples é dada pela expressão:

$$C_{n,p} = \frac{n!}{p!(n-p)!} \quad (1)$$

Portanto, se temos um conjunto \mathbb{V} formado por n vértices tomados p a p , cada combinação gerada será um subgrafo de G contendo p vértices $\in \mathbb{V}$.

1.2. Busca em Largura (BFS)

A busca em largura é um algoritmo para pesquisar em um grafo. Dado um grafo $G = (\mathbb{V}, \mathbb{A})$, e um vértice de origem distinta s , a busca em largura explora sistematicamente G até 'descobrir' cada vértice acessível através de s .

Se temos um conjunto de voluntários $\mathbb{V}' \in \mathbb{V}$, uma Busca em Largura em \mathbb{V}' encontrará os vértices acessíveis a partir de determinada origem. Caso todos os vértices de \mathbb{V}' sejam acessíveis, o grafo induzido por \mathbb{V}' é conexo, visto que G é um grafo não direcionado.

2. Exercício 2 - Algoritmo

Dadas as características do problema, serão descritos os pseudocódigos para ilustrar os principais pontos da solução desenvolvida.

2.1. Leitura do Grafo

Após compilada, a solução deve ser executada através da linhas de comando '\$./executar.sh entrada saída' onde entrada é o nome do arquivo do qual o grafo será lido e saída o nome do arquivo de gravação do resultado, ambos localizados no diretório raíz da aplicação.

O algoritmo 1 implementa o processo de leitura do arquivo de entrada e população dos dados do grafo.

Algorithm 1: Algoritmo para leitura do arquivo do grafo e preenchimento das listas de adjacências e de focos

Data: nome do arquivo através do qual o grafo será lido
Result: Dados do Grafo preenchidos

```
1 index ← 0;
2 values ← ∅;
3 focus ← ∅;
4 line ← first line of the file;
5 countLines ← 1;
6 while line is not empty do
7     values ← line;
8     if countLines = 1 then
9         vertexCount = values[0];
10        edgesCount = values[1];
11        initialize AdjacencyList;
12    else
13        if countLines <= edgesCount + 1 then
14            v1 ← values[0];
15            v2 ← values[1];
16            addEdge(v1, v2);
17        else
18            if countLines = edgesCount + 2 then
19                focusCount ← values[0];
20                initialize focusList;
21            else
22                initialize focus;
23                foreach value ∈ values do
24                    focus ← focus ∪ value;
25                    addFocus(index, focus);
26                    index ← index + 1;
27                end
28            end
29        end
30    end
31    line ← next line of the file;
32    countLines ← countLines + 1;
33 end
34 close file;
```

O arquivo de entrada é aberto e lido linha a linha. A variável *countLines* armazena a linha atual do arquivo. As linhas 9-11 do pseudocódigo indicam a leitura da primeira linha do arquivo, que contém a quantidade de vértices e arestas do grafo, respectivamente. As linhas 14-16 indicam a leitura das arestas, em 19-20 é lida a quantidade de focos e em 24-26 os focos aos quais cada vértice tem acesso.

As adjacências serão armazenadas no mapa de adjacências do grafo e os focos na

lista de focos.

2.2. Combinações

Após a leitura do grafo, são geradas combinações de seus vértices (Algoritmo 2).

Algorithm 2: Algoritmo para geração de combinações dos vértices e retorno da combinação que soluciona o problema ZikaZeroZ

Data: G
Result: Solução do problema ZikaZero

```
1 combinations ← ∅;
2 combination ← ∅;
3 foreach v ∈  $\mathbb{V}$  do
4     combination ← combination ∪ v;
5     if combination contains  $\mathbb{F}$  then
6         return combination;
7     end
8     combinations ← combinations ∪ combination;
9 end
10 for combinationSize ← 2 to countVertices do
11     combinationsAux ← ∅;
12     foreach comb ∈ combinations do
13         if  $|comb| = combinationSize - 1$  then
14             foreach v ∈  $\mathbb{V}$  do
15                 newComb ← ∅;
16                 newComb ← newComb ∪ v;
17                 if newComb ∪  $\mathbb{F}$  and newComb is connected then
18                     return newComb;
19                 end
20                 combinationsAux ← combinationsAux ∪ newComb;
21             end
22         end
23     end
24     combinations ← combinations ∪ combinationsAux;
25 end
26 return ∅;
```

Primeiramente, verifica-se se somente um voluntário será capaz de cobrir todos os focos (3-9). Cada vértice é verificado isoladamente, e se ele possuir acesso a todos os vértices será retornado como solução para o problema. Caso contrário, para *combinationSize* de 2 até $|\mathbb{V}|$, gera-se subconjuntos de \mathbb{V} de tamanho *combinationSize* (10-23), utilizando como base a última combinação gerada (13). Para cada subconjunto gerado, verifica-se se o subconjunto cobre todos os focos e se é um subgrafo conexo de G, caso seja, então o subconjunto é a solução.

Em caso de solução não encontrada, retorna-se um conjunto vazio.

2.3. Focos Acessados

Para verificar se todos os focos são acessados pelos voluntários de determinado subgrafo gerado, o algoritmo 3 é executado, verificando se a quantidade de focos acessados por determinado subgrafo é a mesma quantidade de focos existentes no problema.

Algorithm 3: Algoritmo para verificar se determinado subgrafo possui acesso à todos os focos do problema

Data: subgraph
Result: Se o subgrafo possui acesso à todos os focos

```
1 Seja  $\mathbb{F}$  = conjunto de Focos do grafo original  $coverFocus \leftarrow \emptyset$ ;  
2 foreach  $node \in subgraph$  do  
3   |  $coverFocus \leftarrow coverFocus \cup focus[node]$ ;  
4 end  
5 if  $|coverFocus| = |\mathbb{F}|$  then  
6   | return true;  
7 end  
8 return false;
```

2.4. Conectividade do Subgrafo

Para verificar a conectividade do subgrafo, foi implementado o algoritmo de busca em largura (Algoritmo 4).

Algorithm 4: Algoritmo para executar uma busca em largura no subgrafo e verificar a quantidade de vértices percorridos

Data: subgraph
Result: Quantidade de vértices percorridos durante a busca em largura do subgrafo

```
1 visitedNodes  $\leftarrow \emptyset$ ;  
2  $s \leftarrow subgraph[0]$ ;  
3  $Q = \{s\}$   
4 while  $Q$  not empty do  
5   |  $node \leftarrow RemoveFirst(Q)$ ;  
6   | foreach  $child \in childs[node]$  do  
7     |   |  $visitedNodes \leftarrow visitedNodes \cup child$ ;  
8     |   |  $ENQUEUE(Q, child)$ ;  
9   | end  
10  | end  
11  | return  $|visitedNodes|$ ;
```

O primeiro vértice do subgrafo é escolhido como origem, e seus vértices são acessados primeiro em extensão. Computa-se e retorna-se a quantidade de vértices percorridos durante esta busca.

Após a execução da função, é verificado se $|visitedNodes|$ é igual a $|subgraph|$. Caso seja, o subgrafo é conexo.

O primeiro subgrafo encontrado que contenha todos os focos e seja conexo será retornado como solução para o problema e salvo no arquivo de saída, pois, como são feitas

combinações de menor tamanho até o maior, pode-se garantir que o primeiro resultado encontrado será o melhor (ou de igual tamanho a outros que poderiam ser encontrados posteriormente).

3. Exercício 3 - Análise das Complexidades Temporais e Espaciais

Para a análise da complexidade da solução, considera-se que \mathbb{V} é o conjunto de n vértices (voluntários) e \mathbb{A} é o conjunto de m arestas (laços de amizade entre os voluntários). \mathbb{F} é o conjunto de r focos de reprodução do mosquito.

3.1. Complexidade Temporal

O pior caso da leitura do grafo (Algoritmo 1) tem complexidade $O(A+V*F)$. Essa complexidade é assintoticamente menor que a da computação da solução (Algoritmo 2). Considerou-se, portanto, a complexidade das combinações para a análise.

No melhor caso, o primeiro vértice observado possui, sozinho, acesso à todos os vértices (Algoritmo 2, linhas 3-9). Para esse caso, a complexidade é $O(n)$, visto que não serão geradas combinações e nem será necessário verificar a conectividade, somente se verificará se ele cobre todos os focos, percorrendo-se todos os vértices.

No pior caso da solução proposta, quando a solução for o próprio grafo G , todos os subconjuntos de todos os tamanhos serão gerados. O número de subconjuntos gerados pode ser expresso pela equação 2.

$$C_{n,1} + C_{n,2} + C_{n,3} + \dots + C_{n,n} = \sum_{p=1}^n \frac{n!}{p!(n-p)!} = 2^n = O(2^n) \quad (2)$$

Para cada combinação gerada, primeiramente verifica-se se a mesma possui acesso à todos os focos. Para isso, são percorridos todos os vértices da possível solução (Algoritmo 3, linhas 2-4), para se computar seus focos. A complexidade desta etapa pode ser representada por $O(V)$.

Também será executada uma busca em largura para verificar se o subconjunto encontrado é conexo (Algoritmo 4). Segundo [Cormen 2009], a complexidade assintótica do algoritmo de busca em largura pode ser dada por $O(V + A)$.

Portanto, estima-se que a complexidade da solução desenvolvida seja a representada pela equação 3.

$$O(2^n) * (2V + A) \quad (3)$$

Conclui-se, então, que a solução proposta tem complexidade temporal de ordem exponencial.

3.2. Complexidade Espacial

Visto que para a solução foi utilizado um mapa de adjacências e uma lista de focos, considera-se que mapa de adjacências exija memória de $O(\mathbb{V} + \mathbb{A})$, e a lista de focos

$O(V + F)$. Totalizando, assim, uma complexidade espacial de $O(2V + A + F)$ para o armazenamento do grafo.

Para a geração das combinações, tem-se um conjunto de combinações de tamanho 2^n que é armazenado em memória. Portanto, considera-se uma complexidade espacial de $O(2^n)$.

Visto que a complexidade espacial para armazenamento das combinações é assintoticamente maior que a do armazenamento do grafo, estima-se que a ordem de complexidade desta solução possa ser representada pela equação 4, de ordem exponencial.

$$O(2^n). \quad (4)$$

4. Exercício 5 - Testes da Implementação

Para testar a implementação foram geradas a analisadas instâncias do problema Zika-ZeroZ. Foram geradas instâncias do pior caso, quando a solução é o próprio grafo G, utilizando grafos esparsos, variando a quantidade de vértices de dois a dois, para m igual a ($|A| - 1$) e r igual a $|V|$.

A maior instância testada, no pior caso, para a qual foi produzida solução, possui 18 vértices, 17 arestas e 18 focos.

4.1. Testes de Tempo

Para análise da complexidade de tempo, foram realizados experimentos visando verificar o tempo de execução em função do aumento do número de vértices, arestas e focos.

Os resultados do tempo de execução, em milissegundos, podem ser observados na figura 5 e na Tabela 1.

Tabela 1. Resultado dos experimentos de Tempo

n	m	r	Tempo de Execução (Milissegundos)
2	1	2	15
4	3	4	18
6	5	6	23
8	7	8	109
10	9	10	141
12	11	12	357
14	13	14	1918
16	15	16	24784
18	17	18	304489

4.2. Testes de Espaço

Para análise da complexidade espacial, foram realizados experimentos visando verificar o consumo de memória em função do aumento do número de vértices, arestas e focos.

Os resultados do consumo de memória podem ser observados na figura 6 e na Tabela 2.



Figura 5. Tempo de Execução no pior caso em função do aumento de vértices

Tabela 2. Resultado dos experimentos de Espaço

n	m	r	Consumo de Memória (bytes)
2	1	2	997112
4	3	4	998192
6	5	6	999256
8	7	8	3726648
10	9	10	12588336
12	11	12	24411040
14	13	14	33014056
16	15	16	83734944
18	17	18	208133808

5. Exercício 6 - Comparação de Análise e Execução

Foram realizados experimentos no qual foi avaliado o pior caso da solução, no qual todas as combinações de vértices seriam geradas e apenas o grafo G solucionaria o problema.

5.1. Análise dos Resultados de Complexidade Temporal

De acordo com a previsão teórica descrita na seção 3, a complexidade temporal da solução poderia ser expressa por $O(2^n) * (2V + A)$, equação 3.

Esperava-se, portanto, um crescimento temporal exponencial, em razão do aumento da quantidade de vértices.

Apesar da linearidade do algoritmo de busca e da verificação de cobertura dos focos, como a combinação simples segue uma função exponencial para a quantidade de

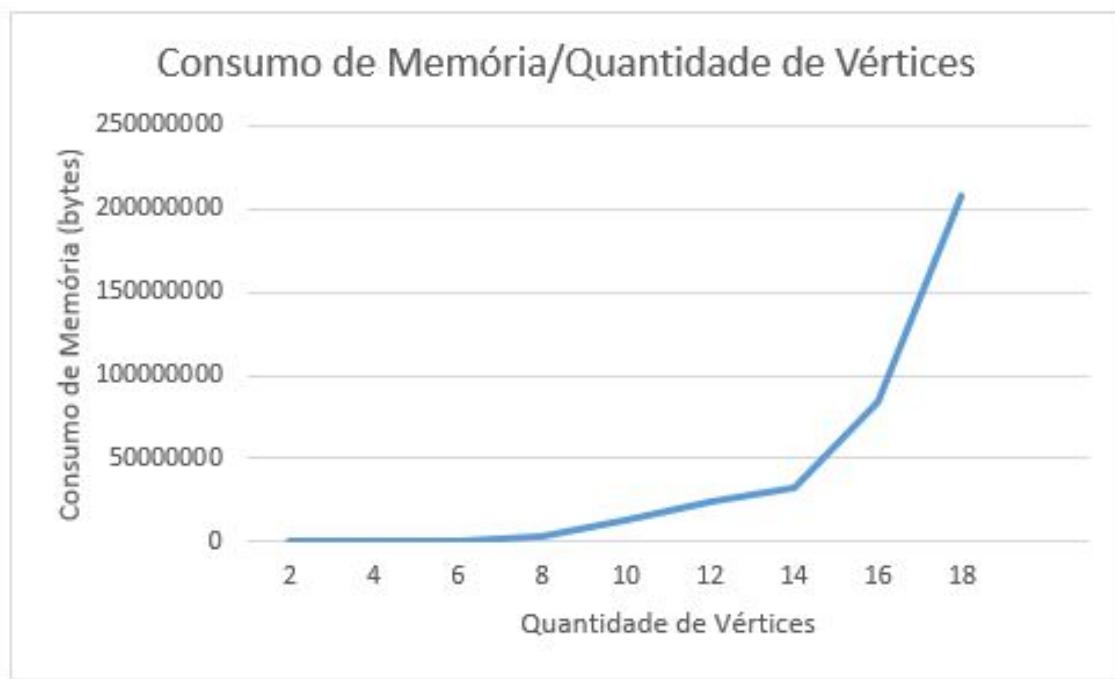


Figura 6. Consumo de Memória no pior caso em função do aumento de vértices

vértices, o que se vê no resultados é uma curva exponencial.

Na figura 5 pode-se observar uma curva exponencial conforme o aumento da quantidade de vértices de G , o que mostra concordância das previsões teóricas com os resultados experimentais.

5.2. Análise dos Resultados de Complexidade Espacial

De acordo com a previsão teórica descrita na seção 3, a complexidade espacial da solução poderia ser expressa por $O(2^n)$, equação 4.

Esperava-se, portanto, um crescimento espacial exponencial, em razão do aumento da quantidade de vértices.

Na figura 6 pode-se observar uma curva exponencial de consumo de memória em razão do aumento da quantidade de vértices de G , o que mostra concordância das previsões teóricas com os resultados experimentais.

6. Conclusão

Este trabalho apresentou uma solução para o problema ZikaZeroZ. A implementação priorizou uma solução ótima em detrimento da eficiência.

Durante os experimentos e análises foi possível perceber que, devido à complexidade exponencial da solução, ela se torna inviável para grafos com grande quantidade de vértices.

Referências

Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.

Trabalho Pratico: ZikaZeroZ

Alan Deivite Guimarães da Silva¹

¹Universidade Federal de Minas Gerais
Av. Antonio Carlos, 6627 - Pampulha - CEP: 31270-010
Belo Horizonte - Minas Gerais, Brazil

{allan.deivite}@gmail.com

Exercício 1. Apresente uma modelagem para o Problema ZikaZeroZ empregando grafos. Deixe claro quais as restrições e/ou suposições devem ser feitas.

Uma modelagem para o problema ZikaZeroZ, que consiste em escolher um grupo dentre diversos voluntários, de modo a formar uma rede de colaboração coesa para eliminar focos de reprodução do mosquito, para ser empregando em um grafo $G(V, E)$, deve considerar o conjunto dos n voluntários como sendo os seus vértices V e os laços de amizades entre os voluntários para a criação de uma rede de colaboração coesa como sendo as suas arestas E , sendo uma restrição para a criação de uma rede de colaboração coesa que o grafo resultante G sejar conexo.

Neste modelo, o grafo $G(V, E)$ é representado por uma lista de adjacência, onde para cada vértice do grafo, uma lista de todos os outros vértices com os quais ele tem uma aresta será armazenada nessa estrutura de dados. Os focos de reprodução do mosquito também será representada por uma estrutura de lista, onde para cada foco, uma lista dos vértices que possuem acesso ao mesmo será armazenada.

Outra restrição que esta modelagem possui para solucionar o objetivo de selecionar o menor número de voluntários $V' \subset V$, tal que, todo foco é acessado, por pelo menos um voluntário $v \in V'$, é a formulação de uma análise combinatória de todos os possíveis vértices, onde o menor conjunto proveniente destas combinações que formarem um grafo conexo e conseguirem cobrir todos os focos preestabelecidos será considerado como sendo a melhor solução possível. A fim de melhorar o desempenho de execução do modelo, a formação do processo de análise combinatória deve ser realizada inicialmente por arranjos unitários, posteriormente por arranjos aos pares e assim sucessivamente, onde para cada arranjo é verificado se uma solução ótima foi atingida. Para critérios de desempate, será escolhido o primeiro arranjo que foi gerado e analisado pelo algoritmo.

Exercício 2. Descreva um algoritmo para resolver o Problema ZikaZeroZ, tendo em vista a modelagem realizada no Exercício 1.

Seguindo a modelagem definida no exercício 1, o algoritmo para solucionar o problema ZikaZeroZ possui as seguintes abordagens:

Primeiramente é realizada a etapa de geração combinatória dos possíveis conjuntos de vértices a serem formados partindo de um grafo inicial $G(V, E)$. O processo de geração dos arranjos é feita de forma crescente, possuindo combinações entre $\{1..|V|\}$ elementos, sendo $|V|$ o número total de vértices.

Para cada conjunto combinatório formado, é efetuada uma inspeção das restrições relacionadas a quantidade de focos que o arranjo analisado consegue cobrir. Cada subconjunto que satisfaz a cobertura de todos os focos tem o seu índice armazenado em uma lista de candidatos a serem os possíveis *minimum subsets*. Posteriormente esta lista passa por uma segunda análise de restrição, sendo excluídos todos os índices que possuem vértices que formam grafos desconexos, sendo adotado o algoritmo *breadth-first search* para verificação de conectividade entre os vértices presentes em cada subconjunto. Ao final das duas análises citadas, se o tamanho da lista de *minimum subsets* for igual a 0, o próximo conjunto combinatório é formado, dessa vez contendo um acréscimo na quantidade de elementos que forma cada subconjunto, caso contrário, significa que a lista de *minimum subsets* possui os índices de todos os subconjuntos com a quantidade mínima de vértices e eles formam um grafo conexo. Para os casos em que a lista de *minimum subsets* apresenta mais de um elemento, apenas o primeiro é considerado adotando como critério o fator deste ter sido gerado primeiramente no processo combinatório.

Exercício 3. Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) do algoritmo proposto no Exercício 2.

O algoritmos proposto no exercício 2 possui como abordagem a formação de todos os conjuntos possíveis de combinações e verificação de qual a menor quantidade de elementos pertencentes a estas combinações que atendem a solução do problema gerando grafos conexos. O pior caso de complexidade temporal para esta abordagem combinatória é da ordem de $O(2^n)$, sendo n o número mínimo de vértices encontrado que soluciona o problema. Para a análise de conectividade, onde o *breadth-first search* é aplicado, as complexidades temporais e espaciais são respectivamente da ordem de $O(|V| + |E|)$ e $O(|V|)$, sendo $|V|$ o número total de vértices e $|E|$ o número total de arestas do grafo. Finalmente para a verificação da restrição de cobertura dos focos pelos vértices, a complexidade espacial é da ordem de $O(|R| + |V|)$, sendo $|R|$ o número total de focos e $|V|$ o número total de vértices que acessam os focos.

Exercício 5. Execute testes da implementação do Exercício 4, propondo instâncias interessantes para o Problema ZikaZeroZ. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n, m e r). Outra sugestão é relatar o tamanho da maior instância para o qual foi produzida uma solução.

Exercício 6. Compare a análise e a execução, respectivamente, Exercícios 3 e 5. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

UFMG – PPGCC/DCC – Projeto e Análise de Algoritmos (PAA)
Primeiro Semestre de 2016 – Tópicos em Grafos
Trabalho Prático: ZikaZeroZ [8 pontos]

Nome: Marcus Rodrigues de Araújo

Matrícula: 2016672310

Exercício 1 [1 ponto]. Apresente uma modelagem para o Problema ZikaZeroZ empregando grafos. Deixe claro quais as restrições e/ou suposições devem ser feitas.

Um grafo, no caso deste exercício, $\mathbf{G} = (\mathbf{X}, \mathbf{Y})$ consiste de um conjunto finito não vazio \mathbf{X} de vértices e um conjunto não vazio \mathbf{Y} de arestas, onde cada aresta é um par não ordenado de vértices distintos. Sejam as seguintes definições:

- \mathbf{V} : um conjunto de vértices representando voluntários que podem eliminar determinados focos do mosquito *Aedes aegypti*;
- \mathbf{F} : o conjunto dos focos do mosquito;
- \mathbf{A} : o conjunto de arestas entre vértices do conjunto \mathbf{V} representando amizade entre essas pessoas;
- \mathbf{E} : um conjunto de arestas entre vértices do conjunto \mathbf{V} e \mathbf{F} ;
- $\mathbf{R}(\mathbf{x})$: como o conjunto de focos acessados pelo conjunto \mathbf{x} de voluntários;

Modelamos o problema ZikaZeroZ, como um grafo $\mathbf{G} = (\mathbf{X}, \mathbf{Y})$ em que:

- $\mathbf{X} = \mathbf{V} \cup \mathbf{F}$;
- $\mathbf{Y} = \mathbf{A} \cup \mathbf{E}$;

Sujeito as seguintes restrições:

- $\mathbf{R}(\mathbf{V}) = \mathbf{F}$;
- \nexists aresta uv tal que $u \in \mathbf{A}$ e $v \in \mathbf{A}$;
- número de vizinhos de qualquer foco f de \mathbf{F} ($f \in \mathbf{F}$) é maior ou igual a 1;

ZikaZeroZ consiste em um problema de se determinar um subconjunto \mathbf{S} de \mathbf{V} ($\mathbf{S} \subseteq \mathbf{V}$) tal que:

- $\mathbf{R}(\mathbf{S}) = \mathbf{F}$;
- \nexists subconjuntos \mathbf{M} e \mathbf{N} de \mathbf{S} ($\mathbf{M} \subseteq \mathbf{S}$ e $\mathbf{N} \subseteq \mathbf{S}$) tal que \nexists caminho entre algum vértice $v_1 \in \mathbf{M}$ e $v_2 \in \mathbf{N}$. Ou seja, o grafo induzido pelos vértices que compõe a resposta deve ser necessariamente conexo;
- \mathbf{S} deve conter a menor quantidade de vértices possível;

Computacionalmente, modelamos o problema da seguinte forma:

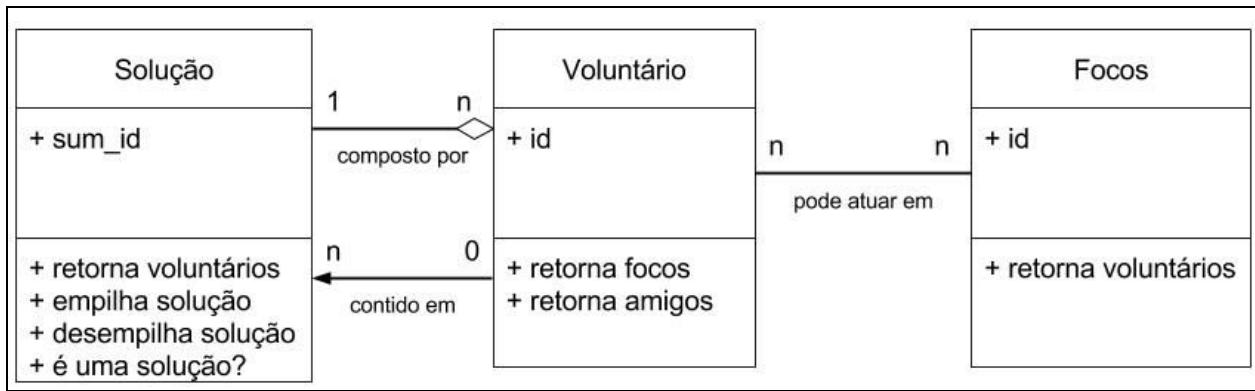


Figura 1. Esquema das principais estruturas de dados utilizadas no trabalho prático

Um exemplo de instância do problema é mostrado abaixo:

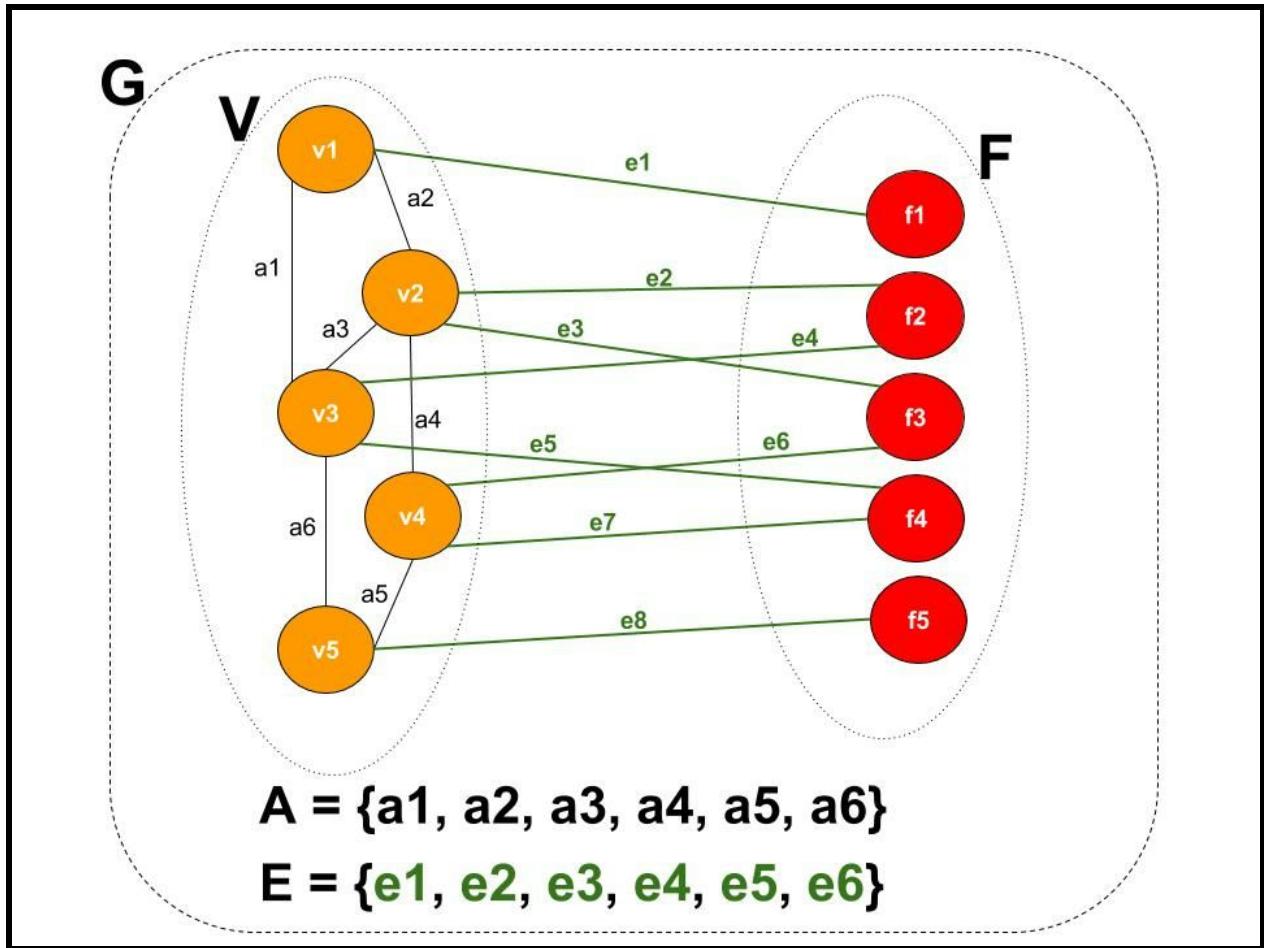


Figura 2. Exemplo de instância do problema

Exercício 2 [5/3 pontos]. Descreva um algoritmo para resolver o Problema ZikaZeroZ, tendo em vista a modelagem realizada no Exercício 1.

2.1) Visão Geral

Para encontrar a solução para o problema, a estratégia utilizada foi gerar combinações de diferentes voluntários k a k , começando com k igual a um e aumentando gradativamente até que se encontrasse um conjunto que cobrisse todos os focos considerados. O algoritmo utilizado pode ser descrito como: faz-se um laço que vai de 1 até o número máximo de voluntários, esse *loop* é responsável por limitar o tamanho das diferentes combinações geradas. No início de cada iteração i , considera-se um conjunto vazio S . Em seguida, verifica-se todas as combinações possíveis escolhendo-se i voluntários. Se na iteração i foi possível encontrar uma ou mais soluções para o problema, elas são retornadas e se escolhe, dentre essas, a que possui a menor soma dos identificadores dos vértices como resposta. Como o algoritmo testa todas as combinações possíveis, de tamanho igual a 1 até p , onde p é o tamanho da menor combinação de voluntários que satisfaz o problema proposto, tem-se a garantia que essa estratégia retorna a solução que seja ótima, ou seja, a menor possível.

No entanto, existe ainda uma restrição a ser satisfeita: "... o grafo induzido por V' (S) em G é conexo". Para garantir que uma solução forme um subgrafo conexo, sempre que se encontra um conjunto de voluntários que cobre todos os focos, testa-se também se o grafo induzido por esses vértices forma um subgrafo conexo.

2.4) Pseudocódigo do algoritmo:

```
// V: grafo dos voluntários e seus laços de amizades
// F: conjunto de focos
// n: número total de voluntários
// k: maneira de selecionar n voluntários
gera_combinações(V, F, n, k):
    soluções : lista<Solução>
    para cada combinação c em combinação(n, k)
        se c cobre_os_focos_em(F)
            se c é_um_subgrafo_conexo_de(V)
                soluções.add(c)
    retorna soluções

// V: grafo dos voluntários e seus laços de amizades
// F: conjunto de focos
procura_por_subconjunto(V, F):
    S : lista<Voluntário> // Solução
    soluções : lista<Solução>
    para i = 1, até i <= V.n // n = número de voluntários
        soluções = gera_combinações(V, F, V.n, i)
        se soluções.quantidade() > 0
            retorna soluções

// V: grafo dos voluntários e seus laços de amizades
// F: conjunto de focos
resolve(V, F):
    soluções = procura_por_subconjunto(V, F)
    solução = soluções.menor_solução()
    retorna solução
```

Exercício 3 [1 ponto]. Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) do algoritmo proposto no exercício 2.

3.1) Complexidade Espacial:

Além do grafo de entrada, a única estrutura de dados que o algoritmo considerado no pseudo código do exercício 2 utiliza é uma lista de soluções. Dessa forma, foi possível analisar a complexidade espacial da solução proposta como:

- Grafo de entrada:

- $G = O(|X| + |Y|)$;
- $X = O(|V| + |F|)$;
- $Y = O(|A| + |E|)$;

- Soluções:

- **soluções** = $O(pq)$, onde p é o tamanho da menor solução possível (quantidade de usuários) e q é a quantidade de soluções encontradas. q pode ser uma fração da combinação de n , p a p , onde n é número total de voluntários.
- **solução** = $O(p)$, em que p é o tamanho da menor solução possível;

3.2) Complexidade Temporal:

```
gera_combinações(V, F, n, k):
    01) soluções : lista<Solução>
    02) para cada combinação c em combinação(n, k)
    03)     se c cobre_os_focos_em(F)
    04)         se c é_um_subgrafo_conexo_de(V)
    05)             soluções.add(c)
    06) retorna soluções

procura_por_subconjunto(V, F):
    07) soluções : lista<Solução>
    08) para i = 1, até i <= V.n // n = número de voluntários
    09)     soluções = gera_combinações(V, F, V.n, i)
    10)     se soluções.quantidade() > 0
    11)         retorna soluções

resolve(V, F):
    12) soluções = procura_por_subconjunto(V, F)
    13) solução = soluções.menor_solução()
    14) retorna solução
```

Sejam as seguintes definições:

n: número de voluntários

k: tamanho do subconjunto de voluntários

r: número de focos considerados

p: menor conjunto de voluntários que formam uma solução para o problema

q: quantidade de soluções encontradas

Temos que as complexidades de cada método do pseudo código são dadas por:

gera_combinações - $O(n!/k!(n-k)! \cdot (kr + k + Adj[k])) = O(n!/k!(n-k)! \cdot kr)$

- 01) declara/inicializa lista de soluções - $O(1)$
- 02) laço é executado $O(p!/(p!(n-p)!))$ vezes
- 03) verifica se os voluntários selecionados cobrem todos os focos - $O(kr)$
- 04) verifica com uma BFS se o grafo formado pelos k voluntários é conexo - $O(k + Adj[k])$
- 05) se essa combinação é uma solução, adiciona-a a lista de soluções $O(1)$
- 06) retorna as soluções obtidas - $O(1)$

procura_por_subconjunto - $O(\sum_{i=1}^p (n!/k!(n-k)! \cdot kr))$

- 07) declara/inicializa lista de soluções - $O(1)$
- 08) laço para gerar diferentes tamanhos de combinações - executado no máximo n vezes
- 09) método para verificar se existe solução envolvendo k voluntários $O(n!/k!(n-k)! \cdot kr)$
- 10) verifica se foi encontrada alguma solução - $O(q)$
- 11) retorna as soluções - $O(1)$

resolve - $O(\sum_{i=1}^p (n!/k!(n-k)! \cdot kr))$

- 12) obtém soluções para o problema - $O(\sum_{i=1}^p (n!/k!(n-k)! \cdot kr))$
- 13.1) seleciona a menor solução - $O(q)$
- 13.2) ordena os voluntários que compõem a solução - $O(p \log(p))$
- 14) retorna a menor solução - $O(1)$

Exercício 4 [5/3 pontos]. Implemente o algoritmo proposto no Exercício 2 na linguagem de programação C, C++, Java ou Python.

Em anexo, a implementação deste trabalho prático em C++ juntamente com a documentação.

Exercício 5 [1 ponto]. Execute testes da implementação do Exercício 4, propondo instâncias interessantes para o Problema ZikaZeroZ. Uma sugestão é que seja produzido um gráfico do tempo de execução por Tamanho da entrada (n , m e r). Outra sugestão é relatar o tamanho da maior instância para o qual foi produzida uma solução.

Os testes realizados foram divididos em grupos. Cada teste foi rodado 6 vezes, sendo que os resultados de tempo foram coletados somente a partir da segunda execução de cada caso de teste, totalizando 5 amostras de tempo para cada entrada. O computador utilizado para rodar os testes possui um processador core i7 com 15 GiB e sistema operacional Ubuntu 12.04 LTS. Em seguida, são apresentados cada um dos grupos de testes assim como informações pertinentes a eles.

Grupo 1: testes fornecidos pelo monitor (inX).

Teste	Voluntários (n)	Amizades (m)	Focos (r)	Atendimentos (e)
in0	5	6	4	7
in1	6	6	6	16
in2	7	6	13	26
in3	8	28	8	8
in4	4	3	4	6
in5	8	21	8	22
in6	5	4	5	12
in7	5	10	8	16

Tabela 1. Informações sobre os testes de entrada (fornecidos pelo monitor)

Teste	Qtd. de voluntários na resposta	Total de Vértices	Total de Arestas	Media (segundos)
in0	3	9	13	0.001
in1	4	12	22	0.001
in2	4	20	32	0.001
in3	8	16	36	0.002
in4	4	8	9	0.001
in5	8	16	43	0.001
in6	2	10	16	0.001
in7	3	13	26	0.001

Tabela 2. Mais informações sobre os testes de entrada (inX) e tempo de execução gasto por cada entrada

Grupo 2: para cada um dos testes do grupo anterior, foi feito com que os grafos de voluntários ficassesem completos, ou seja $m = (n(n-1))/2$.

Teste	Voluntários (n)	Amizades (m)	Focos (r)	Atendimentos (e)
in0'	5	10	4	7
in1'	6	15	6	16
in2'	7	21	13	26
in3'	8	28	8	8
in4'	4	6	4	6
in5'	8	28	8	22
in6'	5	10	5	12
in7'	5	10	8	16

Tabela 3. Informações sobre os testes de entrada (inX modificados)

Teste	Qtd. de voluntários na resposta	Total de Vértices	Total de Arestras	Media (segundos)
in0'	3	9	17	0.001
in1'	3	12	31	0.001
in2'	4	20	47	0.001
in3'	8	16	36	0.002
in4'	2	8	12	0.001
in5'	1	16	50	0.001
in6'	2	10	22	0.001
in7'	3	13	26	0.001

Tabela 4. Mais informações sobre os testes de entrada (inX') e tempo de execução gasto por cada entrada

Grupo 3: foi criado um gerador de casos de teste para se realizar experimentos para esse trabalho. Todos os testes gerados eram instâncias válidas do problema (tinham solução). Abaixo, seguem as informações obtidas ao se submeteras entradas de teste geradas pelo gerador a implementação proposta para este trabalho prático:

Teste	Voluntários (n)	Amizades (m)	Focos (r)	Atendimentos (e)
auto1	10	28	21	51
auto2	15	97	29	83
auto3	16	69	28	100
auto4	17	98	28	90
auto5	28	253	32	162
auto6	20	59	36	132
auto7	32	240	27	170
auto8	32	159	30	192
auto9	30	386	42	200
auto10	32	477	21	145

Tabela 5. Informações sobre os testes de entrada (autoX)

Teste	Qtd. de voluntários na resposta	Total de Vértices	Total de Arestas	Media (segundos)
auto1	4	31	79	0.002
auto2	7	44	180	0.022
auto3	6	44	169	0.019
auto4	7	45	188	0.0544
auto5	7	60	415	2.3554
auto6	8	56	191	0.4206
auto7	6	59	410	1.3658
auto8	6	62	351	1.4442
auto9	8	72	586	15.0976
auto10	4	53	622	0.0346

Tabela 6. Mais informações sobre os testes de entrada (autoX) e tempo de execução gasto por cada entrada

Tamanho da resposta X Tempo de execução

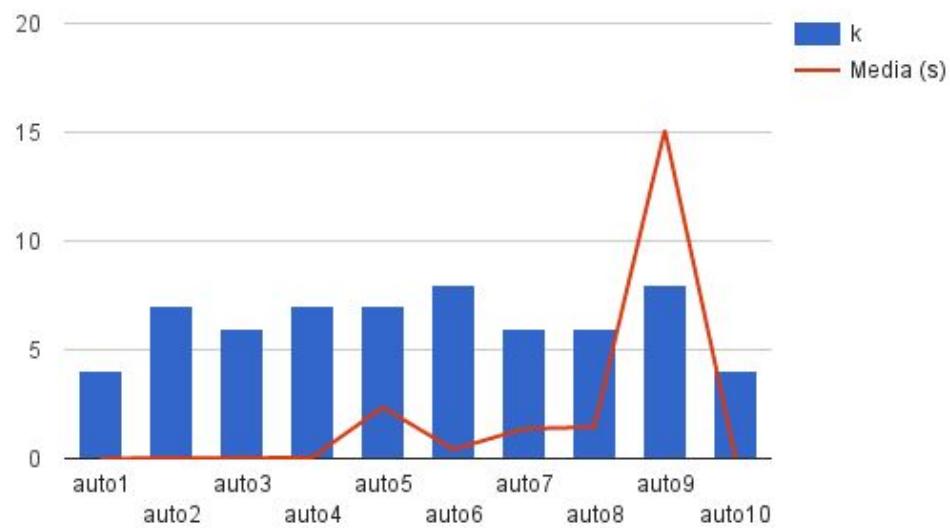


Figura 3. Relação entre tamanho da resposta e tempo de execução. Nesta figura **k** é tido como o tamanho da menor resposta.

Número Total de Vértices X Tempo de Execução

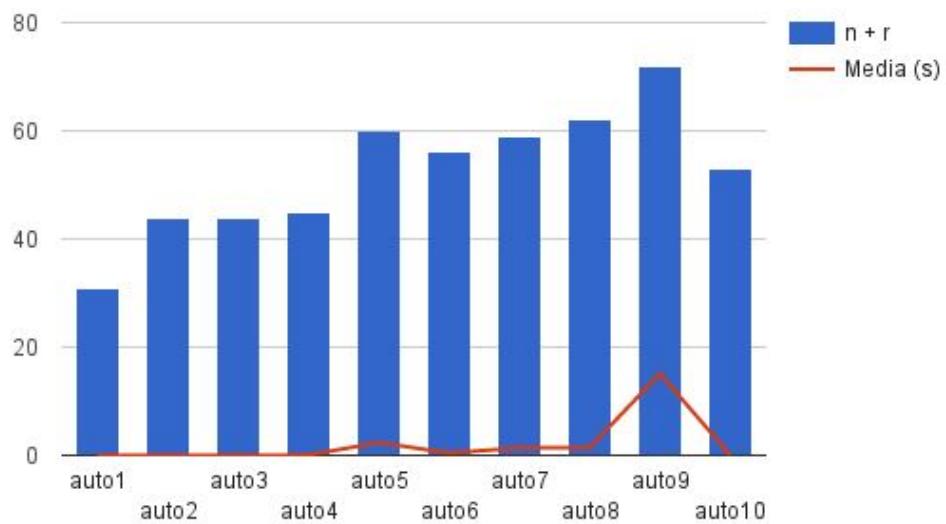


Figura 4. Relação entre número total de vértices e tempo de execução

Número Total de Arestas X Tempo de Execução

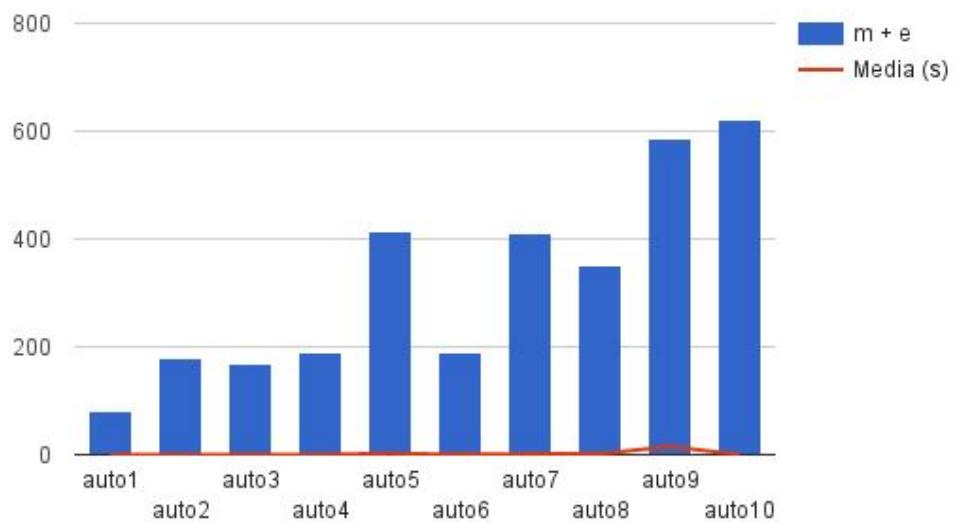


Figura 5. Relação entre número total de arestas e tempo de execução

Grupo 4: maior instância para o qual foi produzida uma resposta

Houveram algumas instâncias geradas pelo gerador de casos de teste que não retornaram solução em um período de 20 minutos. Dentre as instâncias que obtivemos resposta, segue, abaixo, a que dentre elas demorou mais tempo.

Nome: auto13

n = 37

m = 269

r = 57

e = 303

p = 9

Tempo: 6m27.352s

Resposta (voluntários): 1 5 7 13 14 15 24 28 30

Exercício 6 [5/3 pontos]. Compare a análise e a execução, respectivamente, Exercícios 3 e 5. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

Ao se analisar a função de complexidade temporal encontrada para o algoritmo desenvolvido para solucionar o problema proposto neste trabalho prático (TP), temos que ela não se trata de uma função contínua conhecida. Por esse motivo não é tão simples descrever a curva dessa função. No entanto, essas curvas não são necessárias para verificar a dificuldade inerente de se resolver o problema. A diferença na configuração da instância de um problema que gasta 15 segundos para retornar sua resposta e outro que gasta 6 minutos não é muito grande. Quando comparamos as diferenças entre a instância que gasta 6 minutos e outra que não retorna resposta em 20 minutos é ainda menor.

O principal fator complicante para a complexidade do algoritmo proposto aqui é a profundidade para se encontrar uma resposta (quantidade de voluntários na resposta). Isso acontece porque a cada iteração dele mais combinações são geradas. Esse motivo, dificultou a criação de um gerador de testes que explorasse bem isso. Mas, ainda assim, fica claro com os testes feitos o quanto difícil é resolver esse problema computacionalmente. Por fim, é importante lembrar que o problema proposto, como mencionado no fórum, já possui uma natureza fatorial.

TP de Grafos - ZikaZeroZ

Thanis Paiva
Projeto e Análise de Algoritmos

4 de Maio de 2016

Exercício 1. O problema ZikaZeroZ pode ser modelado definindo-se um grafo $\mathcal{G} = (\mathbb{V}, \mathbb{A})$ para representar os voluntários e os laços de amizade entre eles e um conjunto \mathbb{F} de r focos de reprodução do mosquito. A relação entre os n voluntários e os focos acessados por eles é evidenciada pela relação $\mathcal{R}(v) : \mathbb{R} \rightarrow \mathbb{F}$.

Dessa forma, temos um grafo $\mathcal{G} = (\mathbb{V}, \mathbb{A})$ no qual:

- \mathbb{V} corresponde aos nós do grafo, que representam os n voluntários
- \mathbb{A} corresponde as arestas do grafo, que representam os m laços de amizade entre os n voluntários

Temos também um conjunto \mathbb{F} dos r focos de reprodução do mosquito. A relação $\mathcal{R}(v) : \mathbb{R} \rightarrow \mathbb{F}$, explicita os focos aos quais cada voluntário $v \in \mathbb{V}$ tem acesso. Assim, temos uma possível instância para ZikaZeroZ:

$$n = 5 \quad m = 5$$

$$\mathcal{G} = (\mathbb{V} = \{V_1, V_2, V_3, V_4, V_5\}, \mathbb{A} = \{(V_1, V_2), (V_1, V_4), (V_2, V_4), (V_2, V_3), (V_4, V_5)\})$$

$$r = 4$$

$$\mathbb{F} = \{F_1, F_2, F_3, F_4\}$$

$$\mathcal{R} = \{V_1 \rightarrow \{F_1, F_2\}, V_2 \rightarrow \{F_3\}, V_3 \rightarrow \{F_3, F_4\}, V_4 \rightarrow \{F_4\}, V_5 \rightarrow \{F_1, F_2\}\}$$

O objetivo é encontrar o menor número de voluntários, de forma que todos os focos sejam acessados e para qual o grafo induzido $\mathbb{G}' = (\mathbb{V}', \mathbb{A}')$:

- \mathbb{G}' é conexo
- $\mathbb{V}' \subset \mathbb{V}$ é o menor possível, isto é, temos o menor número possível de voluntários
- Todos os \mathbb{F} focos são acessados por, pelo menos, um voluntário $v \in \mathbb{V}'$

Para a instância da Fig. 1, o menor grupo que leva a um grafo conexo e que acessa todos os focos é $\{1, 2\}$. Inicialmente temos $\mathcal{C} = \mathcal{P}(n) = 2^5 = 32$ combinações. Dentre elas, apenas 13

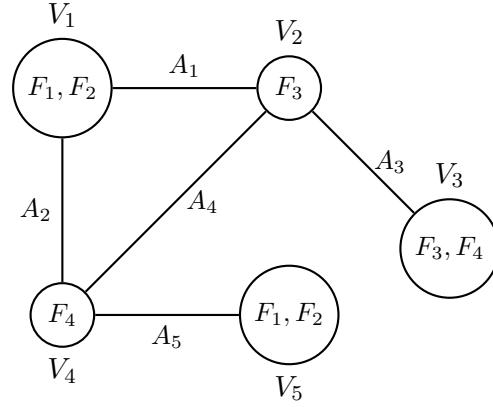


Figura 1: Instância ZikaZeroZ

acessam todos os focos, dentre elas, apenas 9 são conexas e apenas 1 tem o menor número de voluntários, como evidenciado a seguir.

- $\mathcal{C}_1 = \{1, 2\} \rightarrow \mathcal{G}_1$ é **conexo** → tem 2 voluntários → **solução** com menos número de voluntários
- $\mathcal{C}_2 = \{2, 5\} \rightarrow \mathcal{G}_2$ é **não é conexo**
- $\mathcal{C}_3 = \{1, 2, 3\} \rightarrow \mathcal{G}_3$ é **conexo** → tem 3 voluntários
- $\mathcal{C}_4 = \{1, 2, 5\} \rightarrow \mathcal{G}_4$ é **não é conexo**
- $\mathcal{C}_5 = \{2, 3, 5\} \rightarrow \mathcal{G}_5$ é **não é conexo**
- $\mathcal{C}_6 = \{1, 2, 3, 4\} \rightarrow \mathcal{G}_6$ é **conexo** → tem 4 voluntários
- $\mathcal{C}_7 = \{1, 2, 3, 5\} \rightarrow \mathcal{G}_7$ é **não é conexo**
- $\mathcal{C}_8 = \{2, 3, 4, 5\} \rightarrow \mathcal{G}_8$ é **conexo** → tem 4 voluntários
- $\mathcal{C}_9 = \{1, 2, 3, 4, 5\} \rightarrow \mathcal{G}_9$ é **conexo** → tem 5 voluntários

Assim, temos que a **solução** $\mathbb{G}' = (\mathbb{V}', \mathbb{A}')$ é encontrada se e somente se na **entrada** $\mathbb{G} = (\mathbb{V}, \mathbb{A})$, $\mathbb{F} \neq \emptyset$ e $\mathcal{R}(v) : \mathbb{R} \rightarrow \mathbb{F}$, existe **pelo menos um grupo** de voluntários $\mathbb{V}' \subset \mathbb{V}$ no qual:

1. **todos os focos** em $\mathbb{F} \neq \emptyset$ são acessados
2. o grafo induzido $\mathbb{G}' = (\mathbb{V}', \mathbb{A}')$ é **conexo**

Exercício 2. Para encontrar a solução para o problema ZikaZeroZ, foi desenvolvido um algoritmo com quatro passos: (1) gera todas as combinações possíveis de voluntários (sem verificar se existem ou não relacionamentos entre eles), (2) seleciona dentre todas as combinações, aquelas para as quais todos os focos são acessados, (3) seleciona dentre as combinações que acessam todos os focos, aquelas para as quais o grafo induzido é conexo e (4) seleciona dentre os grafos induzidos, o que apresenta menor número de voluntários.

Passo 1 (Linha 1). Gera o Conjunto Potência (*Power Set*) de n , isto é, $\mathcal{P}(n)$, que corresponde a todas as combinações possíveis de n elementos sem repetição e sem importar a ordem dos elementos em subconjuntos com tamanhos de $i = 1 \dots n$, ou seja, temos:

$$\sum_{i=1}^n \binom{n}{i} = 2^n - 1$$

Algorithm 1 Algoritmo ZikaZeroZ

Require: $n \leftarrow \text{numeroVoluntarios}$, $m \leftarrow \text{numeroRelacionamentos}$, $\mathbb{G} = (n, m)$, $f \leftarrow \mathcal{R}(v)$

1: $\mathcal{C} \leftarrow \mathcal{P}(n)$ ▷ Gera o Conjunto Potência de n
2: $\mathcal{C}_{focos} = \{\}$ ▷ Grupos que acessam todos os focos
3: $\mathcal{C}_{conexo} = \{\}$ ▷ Grupos que induzem grafos conexos que acessam todos os focos
4: $S_{zika} = \{\}$ ▷ Menor Subgrafo conexo que acessa todos os focos
5: **for** $i = 1 \dots 2^n - 1$ **do**
6: **if** \mathcal{C}_i acessa todos os focos f **then**
7: $\mathcal{C}_{focos} = \mathcal{C}_{focos} \cup \{\mathcal{C}_i\}$
8: **for all** $C \in \mathcal{C}_{focos}$ **do**
9: **if** C conexo **then**
10: $\mathcal{C}_{conexo} = \mathcal{C}_{conexo} \cup \{C\}$
11: **if** $\mathcal{C}_{conexo}.\text{size}() \neq 1$ **then**
12: $S_{zika} \leftarrow \text{DESEMPATA}(\mathcal{C}_{conexo})$
13: **else**
14: $S_{zika} \leftarrow \text{ORDENA}(\mathcal{C}_{conexo})$
15: **return** S_{zika}
16: **procedure** DESEMPATA(\mathcal{C}_{conexo})
17: **for** $i = 1 \dots \mathcal{C}_{conexo}.\text{size}()$ **do**
18: $labels_1 \leftarrow \text{SOMALABELS}(C_i)$
19: $labels_2 \leftarrow \text{SOMALABELS}(C_{i+1})$
20: **if** $labels_1 == labels_2$ **then return** MENORESLABELS(C_i, C_{i+1})
21: **else**
22: **if** $labels_1 < labels_2$ **then return** C_i
23: **else return** C_{i+1}

elementos em $\mathcal{P}(n)$. Em outras palavras, temos todas as combinações possíveis de voluntários (sem repetição de voluntários ou variação da ordem em que os voluntários são apresentados).

Passo 2 (Linhas 5-7). A partir de todas as combinações $\mathcal{C} = \mathcal{P}(n)$ (**Linha 5**), selecionamos e armazenamos apenas aquelas \mathcal{C}_i nas quais todos os focos $f \in \mathbb{F}$ são acessados (**Linha 6,7**).

Passo 3 (Linhas 8-10). A partir das combinações cujos grupos de voluntários acessam todos os focos, \mathcal{C}_{focos} (**Linha 8**), verificamos quais $\mathcal{C}_i = \mathbb{V}'_i$ correspondem a grafos $\mathcal{G}'_i = (\mathbb{V}'_i, \mathbb{A}'_i)$ que são conexos (**Linha 9**) e os armazenamos em $\mathcal{C}_{conexos}$ (**Linha 10**). Para verificar se o grafo induzido por \mathcal{C}_i , \mathcal{G}'_i é conexo, verificamos se existe um caminho entre cada par de vértices usando uma modificação do algoritmo de *busca em profundidade*.

A *busca em profundidade* foi alterada de forma que dado um par de nós, *origem* e *destino*, a busca é iniciada no nó de *origem* e terminada quando alcançar o nó de *destino*, isto é, encontrou um caminho entre o nó de *origem* e o nó de *destino*. Desse forma, verificamos para cada par de nós da combinação \mathcal{C}_i se eles são alcançáveis, ou seja, se existe um caminho dentro do grafo induzido \mathcal{G}'_i entre cada par de nós da combinação. Se existir um caminho entre cada par de nós, o grafo induzido por essa combinação é conexo.

Passo 4 (Linhas 11-21). A partir das combinações cujos grupos de voluntários acessam todos os focos e o grafo induzido é conexo, \mathcal{C}_{conexo} , selecionamos aquela que apresenta menor número de voluntários (**Linha 12**). Caso o número de voluntários de combinações seja o mesmo, será retornada aquela que apresenta a menor soma dos rótulos dos voluntários (**Linhas 22,23**). Caso o número de voluntários seja o mesmo e a soma dos rótulos também seja a mesma, é retornada a combinação que apresentar o primeiro rótulo distinto que seja menor (**Linha 20**), isto é, comparando $\{1, 4, 5\}$ e $\{1, 3, 6\}$, será retornada a solução $\{1, 3, 6\}$, pois é a que apresenta o menor rótulo ($\{3\}$) que é distinto entre as combinações. A solução encontrada é então ordenada e retornada (**Linhas 14,15**).

Exercício 3. Para fazer a análise de complexidade, consideraremos as seguintes variáveis:

- n = número de voluntários (número de vértices)
- m = número de laços entre voluntários (número de arestas)
- r = número total de focos
- \mathcal{C} = número total de combinações de voluntários
- \mathcal{C}' = número combinações com o mesmo número de voluntários

Exercício 3.1. Análise de Espaço

Inicialmente é criado um grafo $\mathcal{G} = \mathbb{V}, \mathbb{A}$ que apresenta n vértices e m arestas. Esse grafo é implementado na forma de uma matriz de adjacência $n \times n$. Dessa forma, temos um custo de espaço de $\Omega(n^2)$.

Além do grafo $\mathcal{G} = \mathbb{V}, \mathbb{A}$, também é criada uma matriz para armazenar os focos que são acessados por cada voluntário, isto é, a relação $\mathcal{R}(v) : \mathbb{R} \rightarrow \mathbb{F}$. No pior caso, temos que cada um dos n voluntários acessa todos os r focos, de forma que temos uma matriz $r \times n$. Logo, temos um custo de espaço de $O(rn)$.

Além das duas matrizes acima, a que representa o grafo $\mathcal{G} = \mathbb{V}, \mathbb{A}$ e a que representa a relação $\mathcal{R}(v) : \mathbb{R} \rightarrow \mathbb{F}$, temos também listas que armazenam todas as combinações possíveis de vértices. Como cada combinação apresenta ao todo i elementos, com $i = \{1...n\}$, temos ao todo para todas as combinações:

$$\sum_{i=1}^n i \binom{n}{i} = 2^{n-1} n$$

elementos ao todo. Nos passos seguintes essas combinações são reduzidas, primeiro quando selecionamos apenas aquelas para as quais todos os focos são acessados, em seguida a segunda redução ocorre quando selecionamos apenas aquelas combinações que geram grafos induzidos conexos e finalmente temos a redução final que gera uma única combinação que corresponde a solução do problema. Dessa forma, considerando que o custo de espaço para um elemento é $O(1)$, temos um custo de espaço de $O(2^{n-1}n) * O(1) = O(2^{n-1}n)$.

Resumidamente, temos o seguinte custo de espaço: $\Omega(n^2) + O(rn) + O(2^{n-1}n) = O(rn) + O(2^{n-1}n) = \max(rn, 2^{n-1}n) = h(n)$. Assim, temos:

$$h(n) = \max(rn, 2^{n-1}n) = \begin{cases} O(rn) & \text{if } r > 2^{n-1} \\ O(2^{n-1}n) & \text{if } r \leq 2^{n-1} \end{cases}$$

Dessa forma, a complexidade de espaço depende do custo para armazenar para cada vértice e do número de focos acessados. Se o total de focos acessados r for superior a metade do número de combinações $2^{n-1} = \frac{2^n}{2}$, então temos um custo de espaço de $O(rn)$, caso contrário, teremos um custo de espaço de $O(2^{n-1}n)$.

Exercício 3.2. Análise de Tempo

Para analisar a complexidade de tempo, serão analisadas as complexidades de tempo para cada um dos quatro passos do algoritmo e em seguida será feita uma análise agregada para todo o programa.

Passo 1. O cálculo do Conjunto Potência, $\mathcal{P}(n)$, utiliza um contador que registra para cada um dos n voluntários, se o voluntário i faz parte ($\text{contador}[i] = 1$) ou não ($\text{contador}[i] = 0$) de um subconjunto. Logo, para $\mathbb{V}' = \{1, 2, 3\}$, um dos subconjuntos $\{1, 3\}$ seria indicado pelo contador $c = \{1, 0, 1\}$. Como para cada subconjunto o contador de tamanho n deve ser percorrido, temos um custo de $O(n \times (\text{número de subconjuntos}))$.

O número total de subconjuntos, ou $\mathcal{P}(n)$, corresponde a todas as combinações possíveis de n elementos sem repetição e sem importar com a ordem dos elementos em subconjuntos com tamanhos de $i = 1...n$, ou seja, temos:

$$\text{número total de subconjuntos} = \sum_{i=1}^n \binom{n}{i} = 2^n - 1$$

elementos em $\mathcal{P}(n)$. Em outras palavras, temos todas as combinações possíveis de voluntários (sem repetição de voluntários ou variação da ordem em que os voluntários são apresentados).

Como o conjunto potência é gerado a partir de entrada n , em qualquer caso o custo para gerar $\mathcal{P}(n)$ é $O(n \times (\text{número de subconjuntos})) = O(n \times (2^n - 1))$.

Passo 2. Consiste em verificar para cada combinação $\mathcal{C}_i = \mathbb{V}'$ em $\mathcal{C} = \mathcal{P}(n)$, quais são aquelas nas quais todos os r focos são acessados por, pelo menos, um $v \in \mathbb{V}'$. No pior caso, temos a combinação \mathcal{C}_i que apresenta todos os n voluntários, isto é, $\mathcal{C}_i = \mathbb{V}' = \mathbb{V}$ e cada um acessa todos os r focos. No pior caso um voluntário acessa todos os r focos, logo para percorrer a lista de um voluntário temos um custo de $O(r)$. Dessa forma, no pior caso para n voluntários, temos um custo de $n \times O(r) = O(nr)$ para percorrer todas as listas de focos acessados.

Passo 3. Esse passo consiste em verificar se o grafo induzido pela combinação \mathcal{C}_i é conexo. Um grafo induzido não-direcionado é conexo se e somente se existir, pelo menos, um caminho entre cada par de vértices. Logo, verificamos se o grafo induzido por \mathcal{C}_i apresenta, pelo menos, um caminho entre cada par de vértices. Para isso, o algoritmo da *busca em profundidade* foi alterado para retornar todos os caminhos entre cada par de vértices.

O pior caso ocorre quando temos a combinação $\mathcal{C}_i = \{\mathbb{V}\}$, na qual o grafo induzido é o maior possível, ou seja, $\mathcal{G}'_i = \mathcal{G} = (\mathbb{V}, \mathbb{A})$ e \mathcal{G} é completo, ou seja, apresenta uma aresta entre cada par de vértices. Assim, no pior caso temos o número máximo de pares de vértices e o número máximo de caminhos entre cada par de vértices, de forma que a complexidade para o pior caso é dada por:

$$\text{número de pares de vértices} \times \text{custo de todos os caminhos para um par} \quad (1)$$

Número de Pares de Vértices. Como no pior caso o grafo \mathcal{G}'_i induzido pela combinação \mathcal{C}_i é completo, isso implica que existe uma aresta entre cada par de vértices. Como o grafo apresenta uma aresta entre cada par de vértices, temos que o número de arestas m é igual ao número de pares de vértices. Logo:

$$m = \text{número de pares de vértices} \quad (2)$$

Custo de todos os caminhos. Um caminho que se inicia em um vértice de *origem*, no pior caso, visita n vértices até o vértice de *destino*. Para gerar esse caminho, começados no vértice de *origem* como primeiro vértice do caminho, em seguida selecionamos o segundo vértice dentre $(n - 1)$ vértices restantes, o terceiro dentre $(n - 2)$ vértices restantes e assim por diante até chegar ao vértice de *destino*. Assim, temos $(n - 1) \times (n - 2) \times \dots \times 1 = (n - 1)!$ caminhos entre o vértice de *origem* e o de *destino* no pior caso.

Como no pior caso, para um caminho visitamos n vértices, para $(n - 1)!$ caminhos, visitaremos ao todo $n(n - 1)! = n!$ vértices. Dessa forma, o custo no pior caso para calcular todos os caminhos entre um par de vértices é $O(n!)$. Isto é:

$$n \times (n - 1)! = n! = O(n!) \quad (3)$$

Custo da verificação da conectividade. Substituindo (2) e (3) em (1) chegamos ao custo total da verificação da conectividade para um grafo no pior caso calculando todos os caminhos

entre cada par de vértices de $m * O(n!) = O(mn!)$

Passo 4. Consiste em determinar dentre as combinações \mathcal{C}_i que geram grafos induzidos \mathcal{G}' conexos, as que apresentam menor número de vértices. Para isso são calculados o número n de voluntários em cada combinação percorrendo sua lista de voluntários a um custo de $O(n)$. Como temos até \mathcal{C}' combinações que apresentam o mesmo número de vértices, temos um custo total de $\mathcal{C}' * O(n) = O(\mathcal{C}'n)$.

Análise Agregada. Considerando os custos de cada passo, temos um custo total $h(n)$:

$$h(n) = O(n \times (2^n - 1)) + O(nr) + O(mn!) + O(\mathcal{C}'n)$$

Temos que $\mathcal{C} = 2^n - 1$, logo o primeiro termo poderia ser substituído por $O(n\mathcal{C})$. Comparando com $O(\mathcal{C}'n)$, sabemos que $\mathcal{C}' \leq \mathcal{C}$, logo podemos cancelar o último termo, já que o primeiro termo é maior:

$$h(n) = O(n \times (2^n - 1)) + O(nr) + O(mn!) + O(\mathcal{C}'n)$$

$$h(n) = O(n \times (2^n - 1)) + O(nr) + O(mn!)$$

$$h(n) = \max(n \times (2^n - 1), nr, mn!)$$

$$h(n) = O(mn!)$$

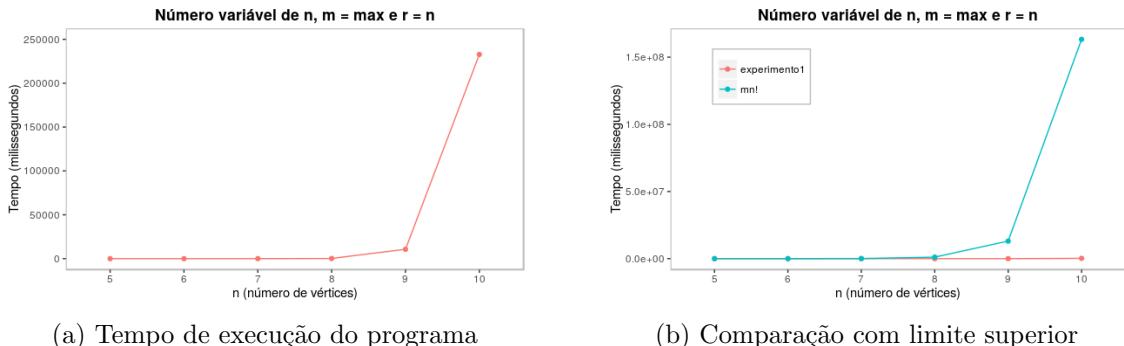
Assim chegamos a **complexidade factorial** de $O(mn!)$. O fator m que multiplica $n!$, indica que o número de arestas, e não apenas o número de vértices influencia o tempo de execução do algoritmo. Assim, para um mesmo número de vértices, o aumento do número de arestas aumentará o tempo de execução do programa. Logo, o aumento do número de vértices e o aumento do número de arestas tem grande influência no tempo de execução. Já o aumento do número r de focos gera um custo linear, de forma que o aumento do número de focos influencia muito pouco o tempo de execução.

Exercício 5. Foram realizados quatro experimentos para avaliar o algoritmo implementado, variando-se as variáveis n , m e r . Todos os tempos de execução correspondem a uma média do tempo em milissegundos de cinco execuções.

Experimento 1: O primeiro experimento consistiu em variar o número de n vértices. O número de focos foi mantido constante em $r = n$ e foram feitos três testes distintos variando-se m . A análise da variação de n e m foi feita em conjunto devido ao limite superior de $O(mn!)$ encontrado na análise teórica. Realizamos três testes, no primeiro, o grafo apresenta o número máximo de arestas, ou seja, $m = \max = \frac{n(n-1)}{2}$ arestas (Fig.2), no segundo temos dois terços do máximo (Fig.3) e um terço do máximo de arestas (Fig.4).

- **Número de arestas $m = max$.** Com o número máximo de arestas, temos um grafo completo, logo, entre cada par de vértices existem $(n - 1)!$ caminhos. Dessa forma, temos um grande aumento no tempo de execução quando aumentamos o número de vértices e colocamos o número máximo de arestas, como podemos observar pela Fig.2a. A maior instância para a qual se produziu uma solução foi para $n = r = 10$ e $m = 45$ (Fig.6a).
- **Número arestas $m = \frac{2}{3}max$.** Quando temos dois terços do número máximo de arestas, temos grafos que são próximos de grafos completos. Com isso, potencialmente também existem muitos caminhos entre os pares de vértices, aumentando assim o tempo de execução, como podemos observar pela Fig.3a. A maior instância para a qual se produziu uma solução foi para $n = r = 12$ e $m = 44$ (Fig.6b).
- **Número arestas $m = \frac{1}{3}max$.** Quando temos um terço do número máximo de arestas, temos grafos mais esparsos, ou seja, são mais distantes de grafos completos. Com isso, há uma redução do número de caminhos possíveis entre os pares de vértices. Assim, temos um aumento do tempo de execução, porém esse aumento é inferior ao dos testes anteriores, como podemos observar pela Fig.4a. A maior instância para a qual se produziu uma solução foi $n = r = 14$ e $m = 30$ (Fig.6c).

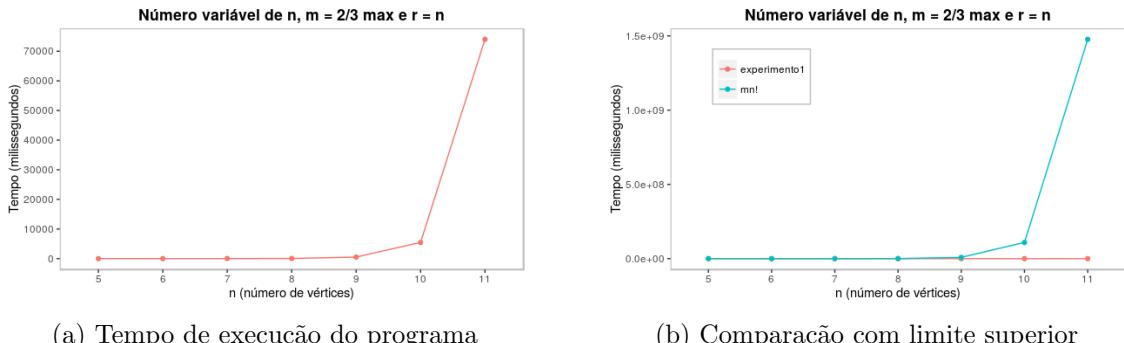
Podemos observar pelos gráficos (Fig.2b,3b,4b) que todas as variações de n e m levam a tempos de execução abaixo do limite superior previsto de $O(mn!)$. Além disso, podemos observar pela Fig.5, que os tempos de execução aumentam com a adição de vértices mais rapidamente quando também aumentamos o número de arestas e o grafo vai se aproximando de um grafo completo. Assim, aumentando o número de vértices e de arestas em conjunto, aumentamos o número de pares de vértices e o número de caminhos entre cada par de vértices até obter o máximo de caminhos, de $(n - 1)!$, em um grafo completo.



(a) Tempo de execução do programa

(b) Comparação com limite superior

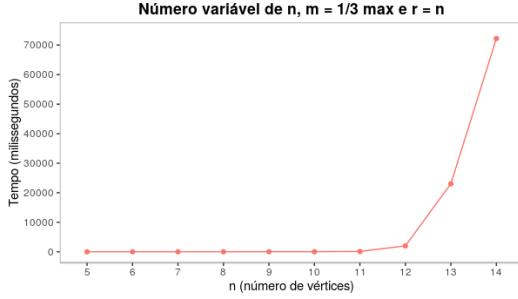
Figura 2: Tempos para n variável, $m = max$ e $r = n$



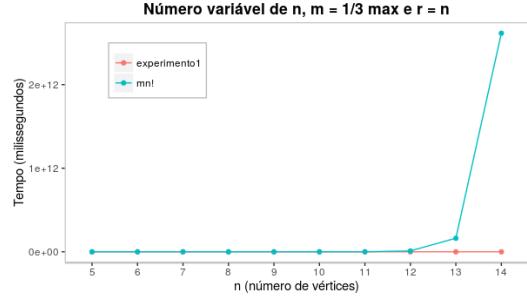
(a) Tempo de execução do programa

(b) Comparação com limite superior

Figura 3: Tempos para n variável, $m = \frac{2}{3}max$ e $r = n$



(a) Tempo de execução do programa



(b) Comparação com limite superior

Figura 4: Tempos para n variável, $m = \frac{1}{3}max$ e $r = n$

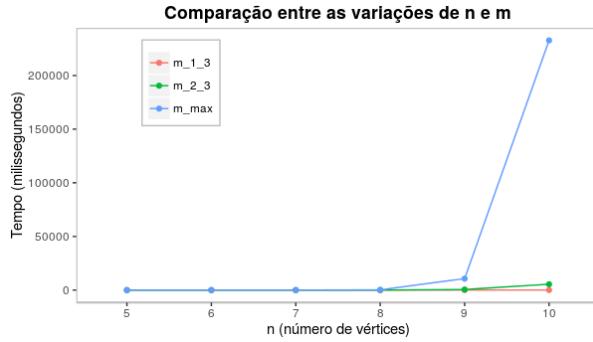


Figura 5: Comparação entre n e $m = \{max, \frac{2}{3}, \frac{1}{3}\}$

Experimento 2: Para o segundo experimento o número de vértices e arestas foi mantido constante em $n = r = 10$, com m variando de 5 até $m = max = 45$ (Fig.7). Assim como no primeiro experimento, chegamos na conclusão de que quanto maior o número de arestas, maior é o tempo de execução para um mesmo número de vértices. Em outras palavras, quanto mais o grafo se aproximar de um grafo completo, maior será o aumento no tempo de execução à medida que adicionamos arestas. Isso ocorre pois quanto maior o número de arestas, maior o número de vértices conectados e consequentemente, maior é o número de caminhos possíveis entre cada par de vértices. Como a geração dos caminhos é a parte de maior custo do programa, o aumento do tempo de execução com o aumento do número de arestas (Fig.7a) é compatível com o esperado e se mantém abaixo do limite superior definido de $O(mn!)$ (Fig.7b).

Experimento 3: No terceiro experimento analisamos a variação do tempo de execução para valores distintos do número de focos r entre $r = 100$ e $r = 2000$, com valores constantes para o número de vértices e o número de arestas, isto é, $n = 10$ e $m = 30$. Existe uma maior variação no tempo de execução para diferentes valores de r (Fig.8a), de forma que valores inferiores, como $r = 300$, apresentam tempos de execução acima dos tempos de execução para valores superiores, como $r = 1000$.

O custo associado ao número de focos r corresponde a verificação da lista de focos acessadas por cada vértice de uma dada combinação, que é linear, $O(nr)$, o que não explicaria essa grande variação. Logo, analisamos os testes criados e observamos que essa grande variação está relacionada com o formato dos arquivos de testes. Os arquivos de teste foram gerados com vértices contendo listas de focos de tamanhos aleatórios, logo, diferentes combinações terão vértices com listas de tamanhos variados, algumas muito curtas e outras muito longas. Essa variação aleatória do tamanho das listas de vértices entre as combinações leva às variações no tempo de execução observadas (Fig.8a).

Entretanto, independentemente da variação do número de focos, esse termo apresenta menor influência no tempo de execução quando comparada com as variações do número de vértices e arestas de um grafo. As maiores instâncias calculadas para n variável apresentavam de $n = 10$ a $n = 14$ vértices (Fig.6). Porém, conseguimos obter resultados para $r = 2000$ em apenas 7 segundos. Logo, podemos concluir que r não influencia tanto o tempo de execução quanto n e m , ficando muito abaixo do limite superior encontrado de $O(mn!)$, como esperado.

Exercício 6. A partir da análise teórica chegamos a um custo fatorial de $O(mn!)$ para o programa e foram feitas as seguintes previsões:

Experimento 1. *O tempo de execução aumenta com o aumento do número de vértices.*

A partir dos testes realizados observamos que o aumento do número de n voluntários, aumenta rapidamente o tempo de execução do algoritmo, o que era esperado, já que ao aumentar o número de vértices aumentamos o número de pares de vértices existentes nas combinações e consequentemente o número de caminhos que devem ser calculados. Além disso, ao aumentarmos o número m de arestas, aumentamos o número de caminhos calculados para cada par de vértices, calculando no pior caso para um par de vértices $(n - 1)!$ caminhos. Logo, nos experimentos percebemos que aumentando n e m aumentamos o tempo de execução do programa, já que aumentamos o número de pares de vértices e de caminhos entre eles. Dessa forma, nossa previsão teórica está compatível com os resultados obtidos.

Outro fato que suporta a nossa previsão teórica de custo fatorial para o programa, é de que foi possível obter soluções apenas para valores pequenos de n , isto é, para no máximo

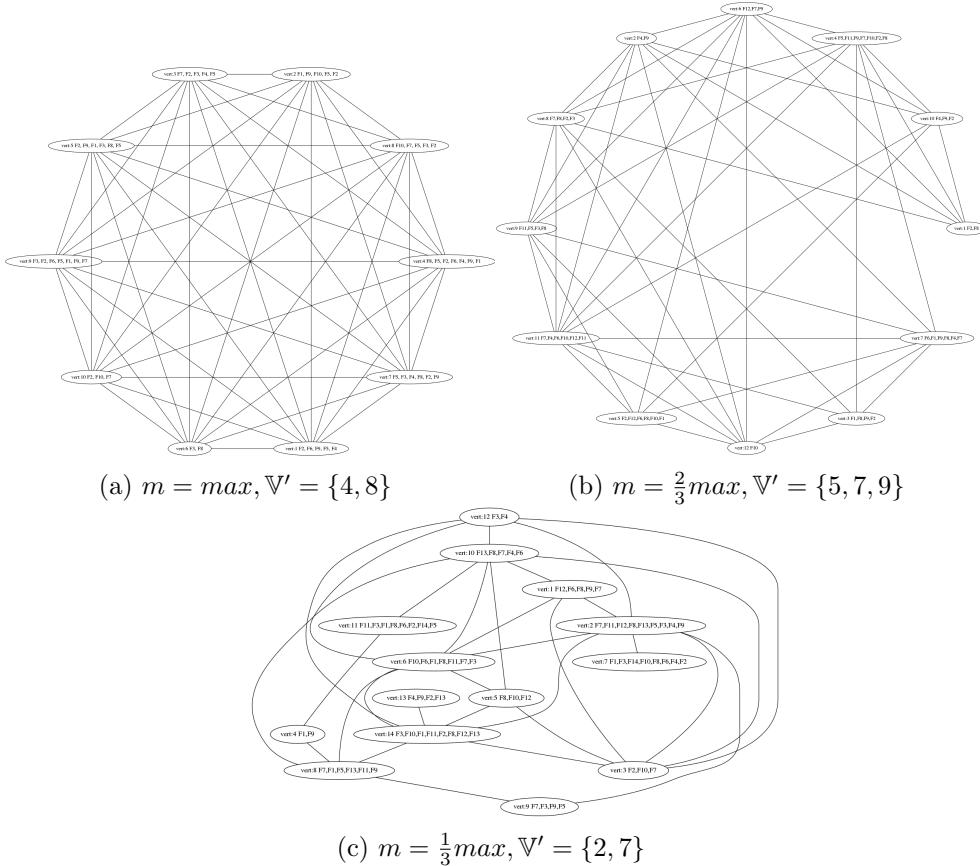


Figura 6: Maiores instâncias para as quais soluções foram produzidas

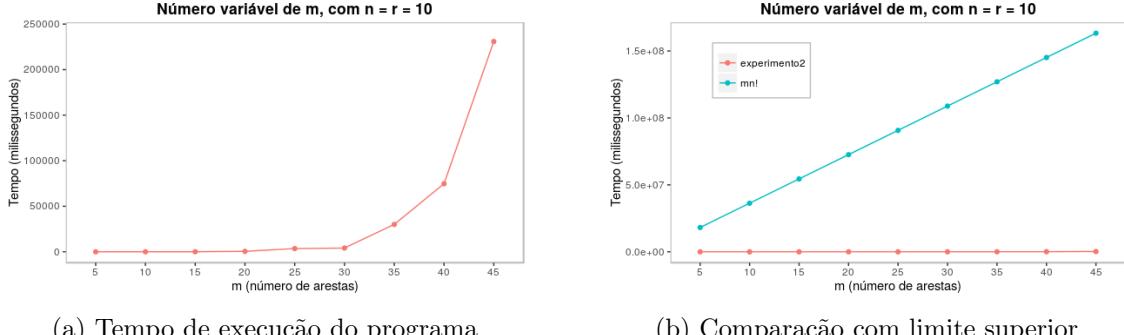


Figura 7: Tempos para m variável, $n = r = 10$

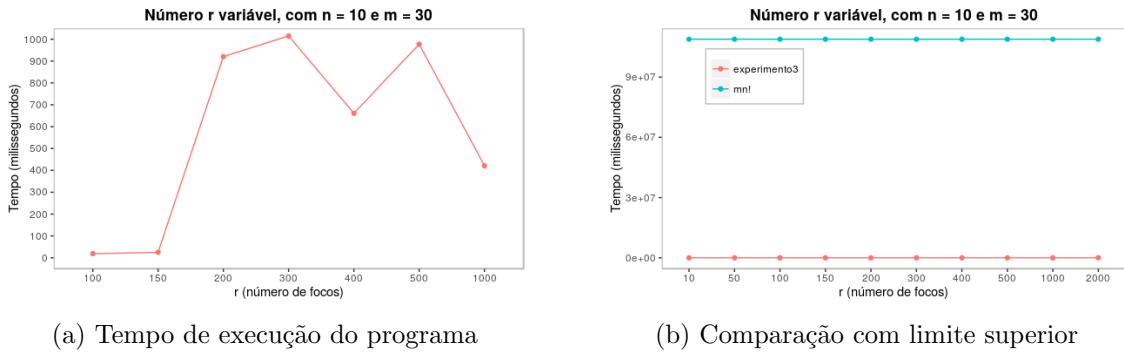


Figura 8: Tempos para r variável, $n = 10$, $m = 30$

$n = 14$, quando o grafo é esparsa (Fig.6c) e $n = 10$ para o grafo completo (Fig.6a).

Experimento 2. *Para um mesmo número de vértices, o aumento do número de arestas aumenta o tempo de execução.*

A partir dos testes realizados, percebemos que o aumento do número m influencia muito o tempo de execução do programa e não apenas o aumento do número de vértices. Para grafos mais esparsos, encontramos soluções para instâncias com até $n = 14$ vértices. Aumentando o número de arestas, reduzimos o número de vértices para os quais o programa encontra uma solução, isto é, com $\frac{2}{3}$ do número máximo de arestas encontramos solução para instâncias com até $n = 12$ vértices. Aumentando o número de arestas até chegarmos em grafos completos, o programa encontra solução para instâncias com até $n = 10$ vértices.

O aumento do número de arestas, assim como o aumento no número de vértices, implica a criação de mais caminhos entre os pares de vértices, aumentando o tempo de execução do programa, como esperado.

Experimento 3. *O aumento do número de focos não tem muita influência no tempo de execução.*

A partir dos testes realizados percebemos que o aumento do número r de focos não afeta o tempo de execução tanto quanto o aumento do número de vértices e de arestas, como esperado, já que está associado a um custo linear de $O(nr)$ para percorrer as listas de focos dos vértices de uma combinação.

As variações no tempo de execução ocorreram devido ao modo como os testes foram gerados. Nos testes, as listas dos vértices apresentam tamanhos aleatórios, provocando variações

do tempo de execução pra diferentes valores de r . Porém, conseguimos gerar soluções para instâncias com até $r = 2000$ focos em apenas 7 segundos, ou seja, mesmo com a variação no tamanho das listas, a influência de r no tempo de execução do programa é muito inferior em comparação com a influência da variação do número de vértices e arestas, o que está compatível com a análise teórica.

Conclusão. Os experimentos produziram resultados compatíveis com as previsões realizadas a partir da análise teórica da complexidade do algoritmo, que apresenta complexidade factorial de $O(mn!)$.

Anexo: Tabelas

Tabelas contendo as informações relativas a execução do programa para os experimentos realizados. Os tempos evidenciados em milissegundos foram obtidos a partir da média de cinco execuções de cada instância testada. A seguir são reportados os valores utilizados na geração dos gráficos usados para a análise experimental da complexidade do programa.

Tabela 1: Dados relativos ao primeiro experimento, com n variável e $m = max$ (Fig.2)

Número de voluntários ($n =$ vértices)	Número de laços ($m =$ arestas)	Número de focos ($r =$ focos)	Tempo de execução (milisseg.)	Tempo limite superior ($m \times n!$)
5	10	5	5	1200
6	15	6	10	10800
7	21	7	26	105840
8	28	8	207	1128960
9	36	9	10671	13063680
10	45	10	232779	1.6e+8

Tabela 2: Dados relativos ao primeiro experimento, com n variável e $m = \frac{2}{3}max$ (Fig.3)

Número de voluntários ($n =$ vértices)	Número de laços ($m =$ arestas)	Número de focos ($r =$ focos)	Tempo de execução (milisseg.)	Tempo limite superior ($m \times n!$)
5	7	5	6	840
6	10	6	13	7200
7	14	7	29	70560
8	19	8	69	766080
9	24	9	525	8709120
10	30	10	5456	1.1e+8
11	37	11	74030	1.5e+9
12	44	12	1501897	2.1e+10

Tabela 3: Dados relativos ao primeiro experimento, com n variável e $m = \frac{1}{3}max$ (Fig.4)

Número de voluntários ($n =$ vértices)	Número de laços ($m =$ arestas)	Número de focos ($r =$ focos)	Tempo de execução (milisseg.)	Tempo limite superior ($m \times n!$)
5	3	5	3	360
6	5	6	3	3600
7	7	7	15	35280
8	9	8	18	362880
9	12	9	26	4354560
10	15	10	46	54432000
11	18	11	111	7.2e+8
12	22	12	2010	1.1e+10
13	26	13	23038	1.6e+11
14	30	14	72226	2.6e+12

Tabela 4: Dados relativos ao segundo experimento, com m variável (Fig.7)

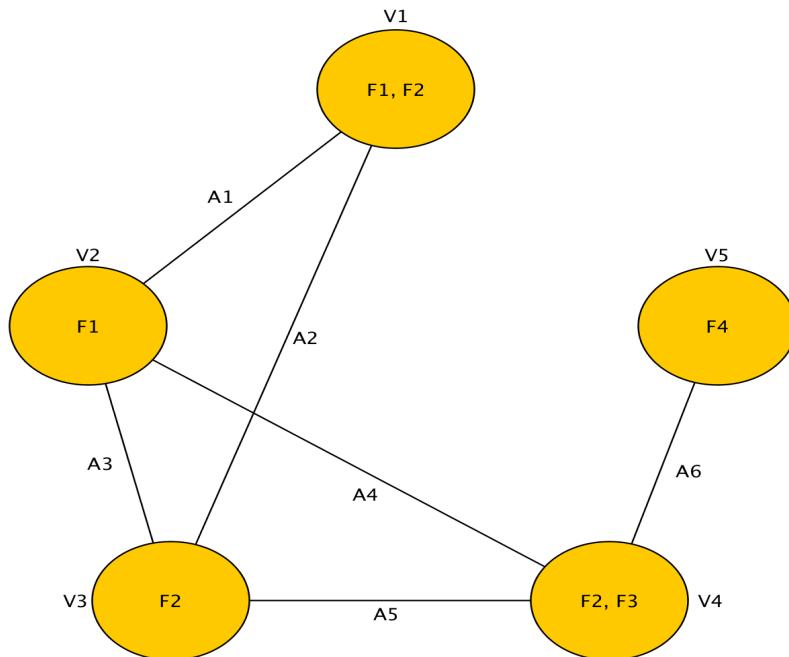
Número de voluntários ($n =$ vértices)	Número de laços ($m =$ arestas)	Número de focos ($r =$ focos)	Tempo de execução (milisseg.)	Tempo limite superior ($m \times n!$)
10	5	10	21	18144000
10	10	10	29	36288000
10	15	10	90	54432000
10	20	10	494	72576000
10	25	10	3560	90720000
10	30	10	4151	108864000
10	35	10	30047	127008000
10	40	10	74737	145152000
10	45	10	230898	163296000

Tabela 5: Dados relativos ao terceiro experimento, com r variável (Fig.8)

Número de voluntários ($n =$ vértices)	Número de laços ($m =$ arestas)	Número de focos ($r =$ focos)	Tempo de execução (milisseg.)	Tempo limite superior ($m \times n!$)
10	30	10	8738	1.1e+8
10	30	50	2652	1.1e+8
10	30	100	19	1.1e+8
10	30	150	25	1.1e+8
10	30	200	920	1.1e+8
10	30	300	1015	1.1e+8
10	30	400	661	1.1e+8
10	30	500	977	1.1e+8
10	30	1000	421	1.1e+8
10	30	2000	6982	1.1e+8

Trabalho Prático de Tópicos de Grafos

Exercício 1. Apresente uma modelagem para o Problema ZikaZeroZ empregando grafos. Deixe claro quais as restrições e/ou suposições devem ser feitas.



- Cada Vértice (V1...Vn) representa um voluntário
- Cada Aresta (A1...An) representa os laços de amizade entre os voluntários
- Um conjunto de focos de reprodução (F) no qual cada voluntário tem acesso
- Suponha que sempre vai existir pelo menos um foco de reprodução em cada vértice
- Suponha que sempre vai existir pelo menos um grafo induzido conexo para cada solução (e.g., V2, V4 e V5 é uma solução e V1, V4 e V5 não).

Exercício 2. Descreva um algoritmo para resolver o Problema ZikaZeroZ, tendo em vista a modelagem realizada no Exercício 1.

```
main()
call GeneratingAdjacencyMatrix(input_file) //for the vertex and edges
call GeneratingAdjacencyMatrix(input_file) //for the focuses
call CopyMatrix(Graph g) //copy everything to a matrix
call BestCase() //test if the best case is the result
call Permutation(int qtElements) //Generate permutations
call PopulateMatrix(ArrayList permutations) //Put the permutations found in a matrix
call TestConnectivity(Solution s) //Test if the subgraph is
```

```

connected from the lowest to the higher

end

function GeneratingAdjacencyMatrix(input_file)
    pick all lines in the input_file and fill
    out the Graph g
    for each line l of input_File
        insert l in G.insertEdge(v, e)
    output Graph g

function CopyMatrix(Graph g)
    pick all elements from the Graph g and send
    to an internal matrix
    for each element g.numVertices and g.numEdges
        insert element in mat[()][]
    output mat[()][]

function BestCase()
    pick all elements in the mat[()][]
    for each element in mat[()][]
        call compareSolution(element)
    output void

function CompareSolution(String[] vertexFocus)
    pick the vertexFocus and test with the solution
    for each sizeFocuses
        solution = 1 //Will generate something like "1111" for 4
elements
    output Boolean meaning the vertexFocus is a solution
    or not

function Permutation(int qtElements)
    pick the qtElements
    set ArrayList p with all possible permutations
    output ArrayList with all permutations based on qtElements

function PopulateMatrix(ArrayList permutations)
    pick all permutations and test they are a solution or not
    for each p permutation of permutation
        call CompareSolution(p)
        if p is a solution
            set ArrayList solution with p
    output matrix with all possible results

function BFS(subGraph s)
    color all the vertices white
    initialize an empty queue Q

```

```

for each neighbor v of s
    insert v in Q
    color[v] = grey
    output s
    color[s] = black
while Q is not empty
    u = top of Q
    remove u from Q
    for each neighbor v of u
        if color[v] = white
            insert v in Q; color[v] = grey
            output u

function TestConnectivity(Solution s)
    pick all the solution s contained in Solution
    for each solution s in Solution
        set Graph subGraph
        call BFS(subGraph)
        if (BFS(subGraph) is true)
            return true
output Boolean representing the solution is connected or not

```

Exercício 3. Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) do algoritmo proposto no Exercício 2.

Complexidade Temporal:

Basicamente, a solução proposta para o problema do ZikaZeroZ precisa de 4 passos primordiais de execução. Dado um grafo $G(n, m)$ onde n é o conjunto de voluntários e m é o conjunto de laços de amizade, e um conjunto r dos focos de reprodução do mosquito: (1) preencher a matriz de adjacência – tanto para a relação entre voluntários e relações de amizade entre os mesmos (n, m), quanto para a relação dos focos com os voluntários, (2) gerar as permutações de soluções, e.g., (a, b, c) é igual a: a, b, c, ab, ac, bc, abc , onde o conjunto vazio é ignorado pela solução, (3) adicionar a solução gerada pela permutação na matriz de soluções e (4) aplicar um algoritmo de busca em largura (BFS) para verificar se a solução gera um grafo conexo ou não.

- Preencher a matriz de adjacência da relação voluntários e laços de amizade (1):

$$O(n^2)$$

- Preencher a matriz de adjacência da relação voluntários e focos de reprodução (1):

$$O(n \times r)$$

- Gerar as possíveis permutações (3):

$$O(2^{n-1})$$

- Verificar se o grafo é conexo através da BFS (4):

$$O(n + m)$$

Pior Caso:

No pior caso de execução do programa proposto para resolver o problema do Zika Vírus, o algoritmo não encontrará uma solução enquanto todas as permutação forem geradas, ou seja, $O(2^{n-1})$, iterações serão geradas antes da solução ser encontrada. Nesse caso específico, somando-se os custos das principais operações do programa (i.e., os 4 passos citados acima), é possível obter o custo temporal do programa no pior caso:

$$O(n^2) + O(n \times r) + O(2^{n-1}) + O(n + m)$$

Melhor Caso:

No melhor caso de execução do ZikaZeroZ, nenhuma permutação do conjunto de vértices será gerada, pois uma rotina foi implementada no código para garantir que todos os casos individuais serão testados antes que qualquer permutação se inicie, e.g., um determinado voluntário n que atende todos os focos de combate ao mosquito é o resultado do problema antes de qualquer processamento posterior. Além disso, o custo da busca em largura (BFS) não será computado, pois somente um voluntário foi capaz de gerar a solução para o problema. Portanto, o custo mais pesado da complexidade temporal não será computado, por essa razão a complexidade no melhor caso é:

$$O(n^2) + O(n \times r)$$

Complexidade Espacial:

A maior quantidade de espaço que o algoritmo vai consumir é no pior caso de execução, onde a matriz de soluções vai receber todas as permutações dos voluntários na execução do

programa. Ou seja, o programa vai gastar:

$$O(2^{n-1})$$

espaço de execução.

Exercício 4. Implemente o algoritmo proposto no Exercício 2 na linguagem de programação C, C++, Java ou Python.

O algoritmo foi implementado na linguagem Java, e se encontra dentro da pasta src com todos os arquivos necessários para testes do mesmo.

Exercício 5. Execute testes da implementação do Exercício 4, propondo instâncias interessantes para o Problema ZikaZeroZ. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n , m e r). Outra sugestão é relatar o tamanho da maior instância para o qual foi produzida uma solução.

O Gráfico 1 mostra o resultado da execução das 8 instâncias propostas pelo monitor da disciplina de PAA. Os dados abaixo mostram o resultado da execução para cada entrada com n , m e r (i.e., n é o número de voluntários, m representa os laços de amizade entre os voluntários e r representa os focos de reprodução que cada voluntário combate). Por exemplo, $in0$ possui $n = 5$, $m = 6$ e $r = 4$. Os tempos foram coletados em milissegundos pelo ZikaZeroZ em Java. É importante ressaltar que cada programa foi executado 5 vezes e depois uma média do tempo de execução foi calculada. Das entradas proposta pelo monitor ($in0..in7$), a que demorou mais tempo para ser executada em milissegundos foi a instância $in3$, que possuía apenas 8 voluntários, mas 28 relações entre eles, além de 8 focos de reprodução atendidos pelos voluntários.

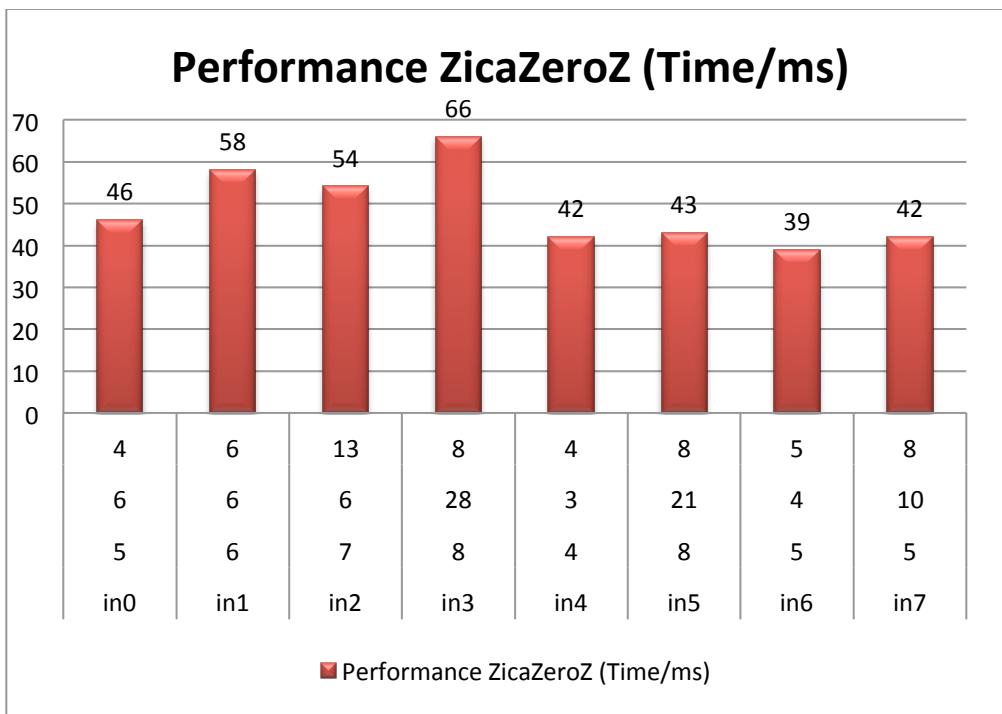


Gráfico 1 - Instâncias propostas pelo monitor da disciplina (*in0..in7*)

Para expandir os resultados encontrados, foi proposto nesse trabalho 3 novas instâncias de testes (*in8*, *in9* e *in10*). O objetivo das 3 instâncias era propor grafos que teriam mais vértices e seriam mais esparsos, procurando atingir o pior caso de execução do algoritmo. A Figura 2 mostra o resultado de execução dos 3 grafos propostos.

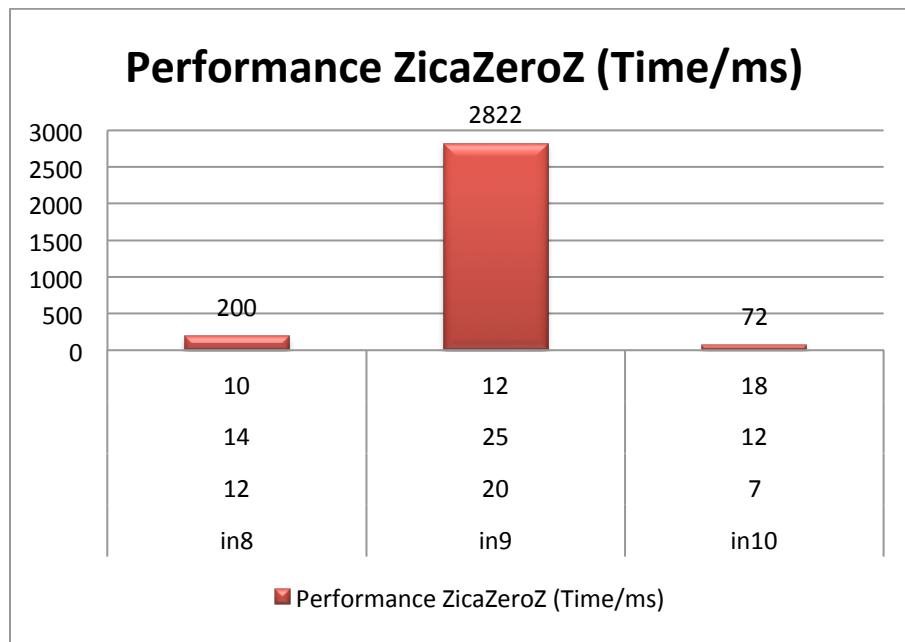


Gráfico 2 - Instâncias propostas pelo autor (*in8*, *in9* e *in10*)

Exercício 6. Compare a análise e a execução, respectivamente, Exercícios 3 e 5. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

A partir da avaliação dos resultados encontrados no exercício 5 com as previsões realizadas no exercício 3, foi percebido que quanto mais esparsa o grafo é, mais tempo o algoritmo vai gastar para finalizar a execução. Ou seja, não importa muito o número de focos que um determinado grafo terá, mas quantos caminhos a permutação vai ter que realizar, antes de encontrar o resultado esperado. Essa suposição vai de acordo com a complexidade temporal do exercício 3, pois tendo em vista o pior caso de execução:

$$O(n^2) + O(n \times r) + O(2^{n-1}) + O(n + m)$$

Pode-se notar que o número de vértices (i.e., n) é a variável que eleva a permutação, além de ter participação em todos os outros fatores do pior caso. Na comparação das instâncias propostas pelo monitor, foi notado que o cálculo do pior caso é correto. A instância *in3*, que representava o pior caso, pois o algoritmo deveria gerar todas as permutações para gerar o resultado, naquele caso (1, 2, 3, 4, 5, 6, 7, 8, ou seja, todos os vértices do grafo). No entanto, o grafo proposto na instância *in3* não era esparsa o suficiente para produzir um resultado muito pior que as outras instâncias. Quando comparado com *in5*, melhor caso de execução, *in3* foi apenas 23 milissegundos mais devagar que *in5*. *In3* e *in5* têm o mesmo número de vértices e o mesmo número de soluções, mas *in3* é o pior caso, além do grafo ser mais esparsa que *in5*.

Para verificar o comportamento exponencial do pior caso, foi usado o mesmo grafo *in3* como base variando os valores de n , m e r . No Gráfico 3 pode-se verificar como o aumento do número de voluntários, laços de amizade e focos de reprodução podem fazer o algoritmo gastar um tempo inviável para valores de n muitos grandes. Como o grafo é inspirado em *in3*, todos os vértices têm caminhos para todos os outros vértices, e cada vértice atende apenas um único foco de reprodução, tornando o grafo o pior caso possível. Para um melhor entendimento do Gráfico 3, todos os 5 grafos propostos foram variando de 2 em 2 vértices começando com 10 até 18. Cada variação de vértices gerava variação de arestas e era proporcional aos focos de reprodução, e.g., *worst_case_10* tem 10 vértices, 45 arestas e 10 focos de reprodução.

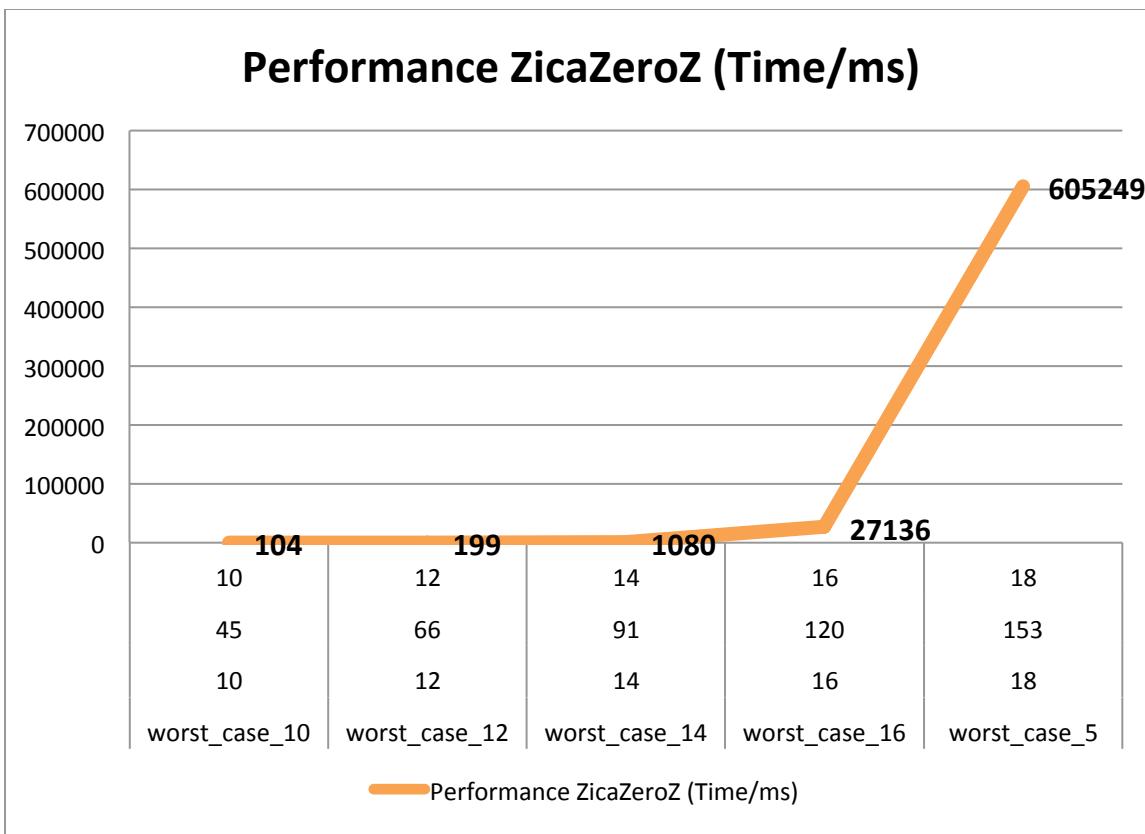


Gráfico 3 - Comparação dos piores casos variando n , m e r

O Gráfico 3 confirma o comportamento exponencial proposto no exercício 3. Uma geração do pior caso para 20 voluntários já era computacionalmente muito custosa para a solução proposta nesse trabalho.

Além disso, com o objetivo de checar como um grafo esparsão poderia fazer o algoritmo demorar muito tempo para executar, foi proposta a instância *in9*, que possuía 20 vértices e 25 conexões entre os vértices (m é muito menor que n^2 , tornando o grafo muito esparsão). Além disso, 12 focos de reprodução deveriam ser atendidos pelo algoritmo. *In9* demorou 2822 milissegundos para achar o conjunto de voluntários, cerca de 43 vezes mais demorado que *in3* (o pior caso proposto pelo monitor). Com isso, foi provado que o número de voluntários e o quão esparsos os voluntários estão uns dos outros é determinante para o desempenho do algoritmo.

Relatório do trabalho prático ZikaZeroZ

Guillermo Ponce Contreras

¹Instituto de Informática – Universidade Federal de Minas Gerais (UFMG)
Caixa Postal 31.270- Belo Horizonte – MG – Brazil

guillermoponce@ufmg.br

Abstract. This meta-paper describes the style to be used in articles and short papers for SBC conferences. For papers in English, you should add just an abstract while for the papers in Portuguese, we also ask for an abstract in Portuguese (“resumo”). In both cases, abstracts should not have more than 10 lines and must be in the first page of the paper.

1. Exercicio

Apresente uma modelagem para o Problema ZikaZeroZ empregando grafos. Deixe claro quais as restrições e ou suposições devem ser feitas.

O problema disse que existe um conjunto de voluntarios para combatir o virus de Zika, a função deles é eliminar os focos de infecção de o virus, e o que tem que fazer é achar o conjunto menor de voluntarios, os quais devem ter um laço de amizade, precisados para accesar a todos os focos. Como é um problema de relações entre objetos, a solução deve estar baseada em grafos.

Então, dado o grafo conexo $G(V, A)$ se sabe:

- V é o conjunto de vértices do grafo, que representa o conjunto de n voluntarios.
- A é o conjunto arestas, que representa o conjunto de laços de amizade entre voluntarios.
- F é um vetor que contém os focos, este vetor representa os focos de infecção que devem ser accesados.
- $V \rightarrow F$ é a relação que existe entre um vértice v e o subconjunto de focos F a os quais ele tem acceso.

Para resolver o problema é necessario ter uma estratégia para percorrer o todos os vértices de V de tal forma que vão ser escolhidos somente os melhores caminhos. Então se pode pensar en usar busca em largura ou busca em profundidade, mudando a condição de parada de o algoritmo, em este caso nao sera quando todos os nos sao visitados, senao quando todos los focos han sido accesados. Na primeira tentativa, se usou uma busca em largura modificada, que vai escolher só o melhor vértice e descarta os outros vértices da lista de adjacencias. Se entende por melhor vértice v a aquele que contém o maior número de focos, de essa forma será possível minimizar el número de vértices neccesarios para a solução ótima. Este processo é repetido v vezes, e em cada iteração se itera a partir de um vértices específico. O problema com essa solução foi que outros caminhos melhores foram descartados, entendendo que é uma estratégia gulosa. A busca em profundidade tem uma ideia similar, assim que também foi descartada.

Em uma segunda tentativa, se pensou em uma estrategia para aumentar o número de caminhos, então se modificou o algoritmo de busca em largura simples mas sem marcar

os vértices como visitados, parando somente se todos os focos foram acessados ou se formam um bucle. Ao final cada caminho será armazenado una estrutura de dados, e se vai escolher a qual tenha menos arestas. Mas essa solução foi também descartada porque por causa de o ordem predeterminado de a lista de adjacencias vai descartar alguns caminhos.

Finalmente, se pensou em maximizar o número de caminhos usando a permutação dos vértices, e de esa forma obter todos os caminhos para escolher o melhor. Então a partir de cada um de os vértices se vao a pegar todos los demais vértices do grafo para e obter todas as permutações a partir de esse vértice. Por exemplo, considerando um grafo com quatro vértices, em função do vértice 1 as possíveis seis permutações são as seguintes: **1** 2 3 4, **1** 2 4 3, **1** 3 2 4, **1** 3 4 2, **1** 4 2 3, **1** 4 3 2. Cada permutação vai gerar uma solução diferente, nenhuma vai ser descartada até acabar com todas as permutações para o grafo, que em total são $n!$ onde n é o número de vértices.

Ao igual que em o algoritmo de busca em largura se vai adicionar o vértice pai a uma fila, e depois em um *loop* vai se pegar o primeiro elemento da fila e adicionar cada um de os vértices adjacentes que não tem sido visitados ainda. A diferença com a ideia proposta e que não se usa a lista de vértices adjacentes, senão o conjunto de vértices complementar de o vértice pai, por um momento se terá esse conjunto de vértices sem considerar a adjacencia, mas em um siguiente paso se vai validar se cada um de os vértices de a lista complementar é ou não e adjacente, só em esse caso vai se passar a o condicional.

2. Exercicio

Descreva um algoritmo para resolver o Problema ZikaZeroZ, tendo em vista a mod- elagem realizada no Exercicio 1.

Como se pode observar no algoritmo 1, na primeira parte de a linha 1 até a 3, se vai executar a função *FIND_PATHS* v vezes, razão e porque *FIND_PATHS*, vai achar todos os caminhos em função de as diferentes permutações geradas sabendo que o primeiro elemento do caminho é o vertice pai v , o qual fica estático. Todas as soluções geradas são armazenadas em uma estrutura, que pode ser um vetor *SLIST* o qual depois vai ser percorrido (linhas 4-9) para achar a melhor solução, que será aquela que tenha menos arestas, e para soluções com o mesmo número de arestas una condição de desempate.

Algorithm 1 SOLVE

Require: G, F

```

1: for each  $v \in V$  do
2:   FIND_PATHS( $v$ )
3: end for
4: for each  $s \in SLIST$  do
5:    $min = \infty$ 
6:   if length( $s$ ) <  $min$  then
7:      $min = \text{length}(s)$ 
8:      $solution = s$ 
9:   end if
10: end for
11: return solution

```

Em o algoritmo 2, sabendo que vai se começar com um vértice passado por parâmetro, que representa o vértice pai ou *root*, vai-se calcular um conjunto complementario de vértices C , que são todos os vértices do grafo sem o vértice pai. De essa forma, se o pai é adjacente a todos os demais vértices, vai-se criar um caminho $SOLUTION = (\text{root} , C[0], C[1], \dots, [C_n])$. Mas como existem casos em que não todos los vértices están conectados uns a os outros, vai-se usar uma função $\text{ARE_CONNECTED}(v, u)$, para avaliar se vértice u esta esta em a lista de adjacencias de v , em outra palavras, se estão conetados, somente em esse caso, o vetor u vai ser visitado. A função ARE_CONNECTED basicamente é uma busca de um elemento em um vetor, então pode ser usada a busca binaria. Se o vértice u está conectado com v e não foi visitado ainda, então vai-se adicionar em a fila Q e será considerado em a solução, depois vão-se seleccionar os focos a os que u tem acceso representados por $\text{Focus}[u]$ e serão adicionados ao vetor de focos LF , o qual vai armazenar todos os focos accesados até o momento (linha 13). O vetor LF é muito importante, porque vai permitir avaliar se todos os focos foram visitados para gerar uma solucão a partir de o caminho avaliado (linhas 20 até 22).

Algorithm 2 FIND-PATHS

Require: $G, root$

```

1:  $C = \text{COMPLEMENT}(root)$ 
2: while  $\text{PERMUTATION}(C)$  do
3:    $\text{INICIALITE\_BFS}(G)$ 
4:    $\text{ENQUEUE}(Q, root)$ 
5:   while  $Q$  is not empty do
6:      $v = \text{DEQUEUE}(Q);$ 
7:      $\text{PUSH}(SOLUTION, v)$ 
8:     for each  $u \in C$  do
9:       if  $\text{ARE\_CONNECTED}(v, u)$  then
10:         if  $\text{color}[v] = \text{WHITE}$  then
11:            $\text{color}[v] = \text{GRAY}$ 
12:            $\text{PUSH}(LF, \text{Focus}[u])$ 
13:            $\text{ENQUEUE}(Q, u)$ 
14:         end if
15:       end if
16:     end for
17:      $\text{color}[u] \leftarrow \text{BLACK}$ 
18:      $\text{PUSH}(FL, \text{focos}[v])$ 
19:     if  $FL.length = \text{FocusNumber}$  then
20:        $\text{PUSH}(SLIST, SOLUTION)$ 
21:     end if
22:   end while
23: end while

```

3. Exercicio

Implemente o algoritmo proposto no Exercicio 2 na linguagem de programação $C, C++, JAVA, PYTHON$.

A implementação está na pasta e foi desenvolvida em c++.

4. Exercicio

Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) do algoritmo proposto no Exercício 2.

Em o algoritmo 1, se sabe por o momento que a primera parte (de a linha 1 até a linha 3) vai ser repetido V vezes o custo de a função $FIND_PATHS$. E na segunda vai percorrer um vetor de $V!$ elementos, que representa todas as possíveis soluções.

Por outro lado, em o algoritmo 2 que representa a função $FIND_PATHS$ vai ter uma complexidade de $O(V^3 \cdot V!)$. Isso as siguiente racões:

- Para calcular o custo de a função $COMPLEMENT$ na linha 1, vai-se percorrer o a lista de vértices uma vez e descartar o vértice conhecido, então vai ter um custo de $V - 1$ que e o mesmo que dizer $\Theta(V)$.
- Depois, o *loop* que começa na linha 2 e termina linha 24, vai ser executado sempre que existir uma permutação de o vetor em questão, em este caso será de o vetor C , então se o vetor tem $O(V)$ elementos o número de permutações posíveis será de $V!$. Mas a função para calcular a permutação tem um custo linear de V .
- Por outro lado, dentro de o *loop* explicado no ponto anterior, se executarão: a) uma remoção de um elemento de uma filha na linha 6, que tem um custo de 1; b) uma adição de um elemento a um vetor na linha 7, que também custo 1; c) e depois vai-se executar V vezes outro *loop* para a verificação de conectividade $ARE_CONNECTED$ (linhas 9-15), que pode ser uma busca sequencial o binaria, considerando que a implementação lineal, o custo é de $O(V)$.
- Fora de o *loop* vai-se verificar se todos os focos foram visitados, essa comparação só tem custo 1.
- Logo, o *loop* de as linhas 8-16, vai ter um custo de $O(V^2)$, por tanto o *loop* de as linhas 5-22, terá um custo de $O(V^3)$ e o *loop* principal de as linhas 2-23 um custo de $O(V^4 \cdot V!)$.

Então o como a função $SOLVE$ executa V vezes a função $FIND_PATHS$, então a complexidade de o algoritmo completo é $O(V^5 \cdot V!)$. O algoritmo algoritmo, não tem um melhor caso, sempre vai-se comportar de a mesma forma, assim e fazer as mesmas comparações, então $\theta(V^5 \cdot V!)$.

5. Exercicio

Execute testes da implementação do Exercício 4, propondo instancias interessantes para o Problema ZikaZeroZ. Uma sugestão e que sea produzido um gráfico do Tempo de execucao por Tamanho da entrada (n,m e r). Outra sugestão e relatar o tamanho da maior instância para o qual foi produzida uma solução.

- Para realizar os testes se usou um computador Sony Vaio Core i3 de 64bits, com 2GB de memoria, além disso o programa foi testado nos computadores do DCC para avaliar que pode ser executado, e executo sem problemas.
- Se testaron todas las entradas brindadas por monitor, e todas foram resoltas sem problemas.
- Se testaram outros exemplos, para avaliar a diferencia entre um grafo esparso e um grafo denso. Em o caso de esparso se colocaram o mesmo número de vértices que de arestas, es também o mesmo número de focos, o grafo também tem uma

distribuição uniforme, por isso cada vértice contém um foco. Se fizeram testes com 2, 3, 4, 5, 6, 7 e 8 vértices, como se pode ver na tabela 1. Como se puede observar na tabela e no gráfico 1, a partir de 7 vértices o algoritmo tem um custo significativo, como se esperava, por outro lado quando se testeó com 9 vértices o computador não pudo fazer e como é um computador um pouco antigo, preferi parar a execução. Estes testes estao na pasta do algoritmo, com os nomes: test0, test1, test2, test3, test4, test5, test6 para os esparsos, e test10, test11, test12, test13, test14, test15, test16 para os densos.

N Vértices	Tiempo (seg) Esparsos	Tiempo (seg) Denso
2	0.000510	0.000525
3	0.001184	0.001480
4	0.006092	0.007993
5	0.023608	0.035634
6	0.219873	0.241583
7	2.250060	2.399910
8	25.51060	48.32540

Table 1. Tabla de comparação de grafo esparso e grafo denso

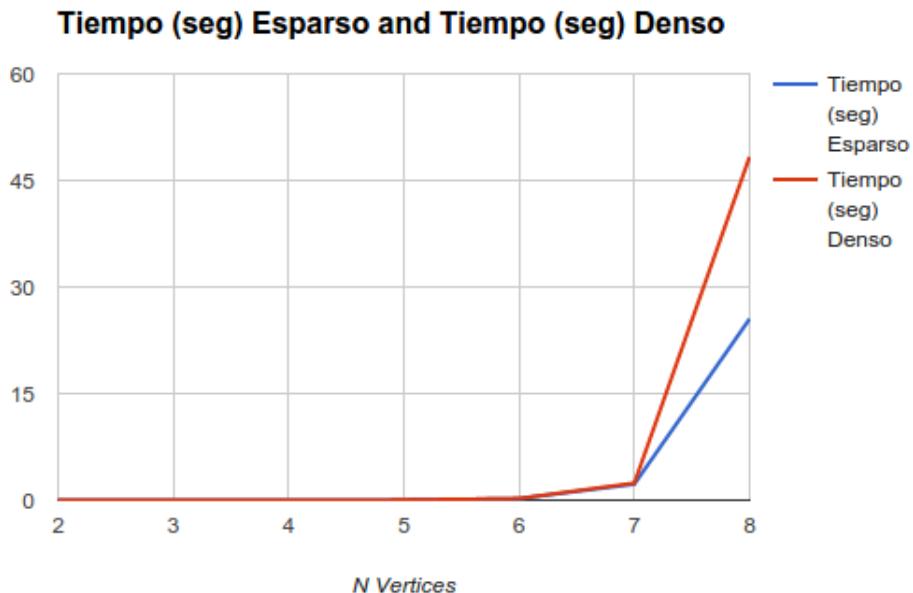


Figure 1. Gráfica de comparação de grafo esparso e grafo denso

- Por outro lado se testaram outros casos, como grafos não conexos, o grafos que contem maior concentração de focos em um so vértice, se resolveram o problema sem problema. Estes testes tambem estam na pasta do algoritmo com nomes: teste20, teste21, teste22.

TRABALHO PRATICO 1, PAA, GRAFOS

Yuri Pessoa Avelar Macedo - 2016662047

Observações

Para todas as questões abaixo, utiliza-se as seguintes definições:

$G = \text{graph (grafo)}$
 $V = \text{vertices}$
 $E = \text{edges (arestas)}$
 $R = \text{focus (focos)}$

Conforme definido pelo enunciado, o relatório e documentação deste trabalho prático deve seguir de forma sucinta e objetiva, tal e como em uma prova escrita cujos tópicos são as questões enumeradas pelo enunciado. Portanto, esta documentação seguirá um modelo semelhante a de prova, sem a inclusão de seções descritivas como sumário, referencias, resultados, conclusão, etc. Cada seção foca objetivamente a seu respectivo tópico.

Questão 1 Modelagem

Dado um grafo G não orientado e não ponderado, em que cada vértice possui de 1 a R focos de um conjunto de R focos. O problema, conforme mencionado no enunciado, envolve encontrar o menor conjunto G^1 de vértices de G tal que todos os vértices de G^1 são alcançáveis por qualquer outro vértice de G^1 . Também é exigido que G^1 contenha todos os focos do conjunto R .

Comparando a outros problemas, é possível perceber que o sistema formado é bastante complexo no sentido que é difícil determinar se, a cada momento, um vértice de V é garantido ou não de existir em uma solução ótima G^1 . Não há uma estratégia gulosa que é capaz de definir um ponto de partida, e não há uma garantia de que um dado vértice pertence à solução ótima sem antes comparar todos os demais caminhos.

Mesmo sem uma definição formal, ou uma prova por redução, assume-se que o problema pertence a uma classe NP-Hard ou NP-Completo. Esses problemas, até onde se é conhecido, não são possíveis de serem resolvidos em tempo polinomial sempre retornando uma solução ótima. Heurísticas que são usadas para resolver esse problema não garantem uma solução ótima, o que vai contra a exigência do enunciado. Portanto, assume-se que a estratégia utilizada para resolver o problema deve ter complexidade da ordem de $O(N!)$ ou de $O(2^n)$. Esse tipo de solução sempre será ótima, mas terá de ter péssima escalabilidade para problemas grandes.

Na definição do algoritmo que se segue na Questão 2, assume-se que:

1. *Nenhum vértice pode possuir 0 focos.*
2. *No arquivo de entrada, cuja formatação é descritameticulosamente no enunciado, não há repetições de arestas (u, v) e (v, u)*
3. *Embora o arquivo de saída exige que os vértices da solução ótima estejam separados por espaço, pode haver um espaço adicional após o ultimo vértice da saída.*
4. *O critério de desempate para duas soluções ótimas é a ordem crescente do primeiro até o ultimo vértice da solução.*
5. *Assume-se que existe pelo menos uma solução G^1 tal que essa contém todos os vértices de G e essa pode ou não ser ótima.*

Já planejando para uma implementação em C++, define-se que o grafo será representado por um **vector** (estrutura de C++ que assemelha-se a uma lista encadeada) de tamanho V que contem objetos de uma classe **vertex** que representa os vértices do grafo. Cada **vertex** v possui outros dois vectors. Um deles contem uma lista de inteiros com as chaves de cada vértice em G para o qual dado vértice v é adjacente. O outro contem uma lista booleana de tamanho R que contem $r[x] = \text{true}$ caso o vértice v possua o foco de número x , e $r[x] = \text{false}$ caso o vértice v não possua o foco de número x .

Questão 2 Algoritmo

Dado que para obtermos uma solução ótima, temos de observar todos os caminhos, o algoritmo deve ser capaz de percorrer todas as soluções para G^1 idealmente uma única vez. Entretanto, dado um subgrafo G^2 de G não é possível garantir de forma

trivial que este possui todos os focos e é conexo, sem custo adicional durante sua construção. A não ser que o grafo seja construído por percorrer uma árvore a partir de todos os nós, o que gastaria tempo $O(V^2 + VE)$ com muitos resultados sendo redundantes, não é possível garantir de forma linear que o Grafo V' é conexo durante sua montagem.

Sendo assim, optou-se pela seguinte estratégia: montar todos os subgrafos de G exatamente uma vez, e verificar se eles atendem as condições que os tornam soluções ótimas G^1 . Essa montagem é realizada na seguinte ordem para as verificações pelas quais procura-se atender a $G^2 = G^1$

1. Montagem do gráfo G^2 . Com tamanho de G de 1 vértice até V vértices
2. Verificação por todos os Focos
3. Verificação de Conectividade
 - Se passar, solução ótima encontrada
 - Se falhar, retorne a 1.

A montagem do grafo G^2 envolve a criação de todos os subgrafos de G , independente da satisfação das condições de focos e conectividade do grafo. Por exemplo, se um grafo possui, 5 vértices, o algoritmo todas as combinações possíveis de 1 vértice (1 - 2 - 3 - 4 - 5). Em seguida, ele cria todas as combinações de dois vértices (1,2 - 1,3 - 1,4 - 1,5 - 2,3 - 2,4 - 2,5 - 3,4 - 3,5 - 4,5), e assim por diante até que uma solução seja encontrada ou todos os subgrafos possíveis sejam construídos. O número de combinações que serão criadas é definido combinatorialmente sobre V e o número i de vértices de G^2 . Isso será melhor definido na Questão 3.

Algo importante sobre a montagem envolve escolher se os grafos a serem testados serão construídos na ordem de tamanho de 1 até V vértices ou de V até 1 vértices. Algumas das vantagens que nos faz optar por montar pela primeira opção envolve a garantia de que a primeira solução encontrada é sempre ótima, uma vez que o algoritmo até o momento teria encontrado, para o menor número de vértices possível, apenas soluções invalidas. Outra vantagem é que muitas das complexidades, conforme será visto na Questão 3, dependem principalmente de V . Isso faz com que a escolha de iniciar por grafos com V menores uma heurística eficiente para reduzir o tempo de execução.

A verificação dos focos envolve, após todos os vértices de G^2 serem selecionados, garantir que não há nenhum foco de R que não está presente no G^2 . Essa verificação é feita em tempo R dado a estrutura do vector de focos presente em cada vértice. Essa verificação é feita antes da verificação por conectividade por ser, para a maioria dos problemas, mais barata.

A verificação de conectividade envolve, dado o grafo G^2 verificar se ele é conexo. Essa é feita de forma linear por meio do algoritmo de busca em profundidade (DFS) visto durante o curso. Nele, uma função *DFS* chama a uma função recursiva *DFS_Recursion* que percorre o grafo. Se depois de percorrer todos os caminhos do grafo a partir de um vértice arbitrário v , não foi possível chegar a todos os vértices, a função *DFS* possui um loop que inicia uma nova busca a partir de vértices ainda brancos (*não encontrados*). No caso deste problema, quando isso ocorre, temos certeza de que o grafo não é conexo e portanto não atende às condições que o classificam como uma solução ótima G^1 . Essa verificação custa $O(V+E)$, sendo quase sempre mais cara que a verificação por foco.

Questão 3 Complexidade de Tempo e Espaço

Ao que se refere a complexidade de espaço, o algoritmo opera somente sobre o grafo original na forma de vector, com cada vértice possuindo 2 outros vectors: Sua lista de adjacências, e sua lista de focos. O número de adjacências total pode ir de $E = V$ para grafos esparsos até $E = 2V^2$ para um grafo denso (*pior caso*), uma vez que se a aresta $E(u,v)$ existe no grafo, então tanto u como v a possuem em suas listas. Já a lista de focos pode ir 1 até R para cada v em grafos nos quais cada vértice possui muitos, se não todos, dos R focos.

- **Sendo assim, a complexidade final de espaço é dada por** $\Theta(VR + E)$
- **Melhor Caso (grafo esparsão com 1 foco):** $\Omega(2V) = \Omega(V)$
- **Pior Caso (grafo denso com R = V):** $O(VR + 2V^2) = O(3V^2) = O(V^2)$

Ao que se refere à complexidade de tempo, temos que, para um conjunto de $N = V$ vértices, obter todas as combinações possíveis de uma seleção de i vértices sem repetição é definido combinatorialmente por $\frac{N!}{i! * (N-i)!}$. Como estamos interessados em todos os i de 1 (*melhor caso*) até N (*pior caso*) vértices, definimos o somatório da complexidade nesse intervalo como $\sum_{i=1}^N \frac{N!}{i! * (N-i)!}$. Embora a complexidade já seja alta dado a natureza de força bruta do algoritmo, ainda é necessário verificar que:

para qualquer conjunto de i vértices de N , o conjunto possui todos os focos (custo $\Theta(R)$) e que o conjunto é conexo, que é feita de forma linear com uma variação do algoritmo de busca em profundidade $\Theta(V+E)$. Como $V = N$, temos que:

- **A complexidade final de tempo é dada por** $\Theta(R) * \Theta(V+E) * \sum_{i=1}^V \frac{V!}{i!(V-i)!} = \Theta(2^V R (V+E))$
- **Melhor Caso (Quando um vértice contém todos os focos, $R = 1$, $E = V$):** $\Omega(V(V+E)) = \Omega(2V^2) = \Omega(V^2)$
- **Pior Caso ($R = V$ (mesmo que R não tenha limite), grafo denso com $E = V^2$, e cada vértice possui um foco):** $O(2^V V(V+E)) = O(2^{V+1} V^2)$

É importante reforçar que os melhores e piores casos para a complexidade de tempo e espaço não são necessariamente os mesmos cenários, podendo ter uma relação até inversamente proporcional. Quanto mais conectado o grafo, maior é a complexidade de espaço, e menor é a dificuldade de se conectar um menor número de vértices. Quando os vértices possuem poucos focos, maior é a complexidade de tempo, e menor é a complexidade de espaço.

Questão 4

Implementação

O código referente à implementação em C++ do algoritmo para resolver o problema do Zika vírus está em anexo a este documento, seguindo os padrões de compilação e execução exigidos no enunciado.

Questão 5

Análise da Implementação

Dos testes para garantir o funcionamento correto do algoritmo foram utilizados a entrada modelo oferecida pelo enunciado além de outras 6 entradas fornecidas pelos forums do moodle contendo as respostas ótimas. Para todos esses testes, o algoritmo foi capaz de oferecer a mesma solução ótima que as fornecidas. Vários cenários interessantes puderam ser testados por meio dessas entradas, incluindo:

1. Grafo conectado cujos todos menos um vértices estão conectados a um único vértice v , sendo que esse vértice v está conectado a todos os outros.
2. Instancia de grafo denso fortemente conectado.
3. Instancia de grafo denso fortemente conectado, no qual cada foco está presente em apenas um dos vértices. (pior caso)
4. Grafo acíclico em linha.
5. Grafo G não conexo. Esta instância prova que G não deve ser necessariamente conexo para que G^1 seja conexo.

Para testar tempo de execução, planejou-se a análise de complexidade de tempo do melhor e pior caso, que são definidas pelas condições descritas na Questão 3. Percebeu-se que no verdadeiro melhor caso, que é quando a solução está no primeiro vértice encontrado, o tempo de execução torna-se constante, uma vez que a solução é encontrada em uma iteração apenas, e o algoritmo deixa de executar. Portanto, para o melhor caso abaixo, considera-se que o grafo possui $R = 2$ e que o conjunto ótimo requer pelo menos uma verificação de todos os vértices (um dos focos está localizado sempre no último vértice verificado, então a solução será sempre o primeiro e o último vértice do grafo). Para testar o pior caso, cria-se um grafo denso cuja solução sempre será encontrada no pior caso (a solução será sempre todos os vértices do grafo):



Grafico de "Número de vértices" x "Segundos" para pior caso, melhor caso com no mínimo V análises.

Entrada Base	0.001 Seg
in1	0.001 Seg
in2	0.003 Seg
in3	0.007 Seg
in4	0.000 Seg
in5	0.002 Seg
in6	0.002 Seg

As entradas in1 a in6 referem-se às entradas utilizadas para testar o funcionamento correto do algoritmo. Essas entradas são as mesmas que as fornecidas no forum de discussões do Moodle pelo monitor. Assume-se que a explicação de cada entrada "in" não é necessária.

Os testes foram realizados de dois em dois vértices adicionados de $V = 2$ até $V = 20$. No melhor caso, com o número mínimo de vértices verificados sendo V , percebeu-se pouco crescimento. Até 12 vértices, o algoritmo consome apenas 0,005 segundos. Mesmo em 20 vértices, o crescimento continua insignificante, consumindo 1,517 segundos.

Para o pior caso, a característica de péssima escalabilidade do algoritmo torna-se aparente, com os pulos entre 14, 16, 18 e 20 vértices sendo 0,702, 3,27, 15,086, 68,145 segundos, respectivamente. A maior instância testada foi a de 20 vértices para o pior caso. Visto o pulo entre 18 e 20 vértices, o esperado é que o aumento torne-se cada vez maior.

Essas instâncias do pior caso foram baseadas na entrada “in3” fornecida nos forums. Foi curioso que para 6 e 8 vértices, o algoritmo executou em 0,001 e 0,008 segundos, respectivamente. Mas para in3, que é exatamente o mesmo problema com 7 vértices, o tempo de execução foi de 0,007, o que confirma as expectativas quanto às configurações do pior caso.

Questão 6 Comparação com o Esperado

Em questões de escalabilidade, o algoritmo se comportou exatamente como o esperado. Testes um pouco maiores, e o programa levaria minutos para realizar a computação de um grafo. Para o melhor caso, foi uma surpresa que as contingências heurísticas como a ordem de verificação e o crescimento do tamanho do grafo ajudam bastante a filtrar grafos G^2 com poucas probabilidades de serem ótimos.

O tempo de execução também conforme esperado é extremamente dependente do número de vértices. Mesmo com outros parâmetros mais altos como em “in3” com $R = V + E$, o tempo de execução ainda é pouquíssimo influenciado, próximo a instâncias que possuem muitos vértices e muitas arestas. Quanto maior o numero de arestas, mais próximo de $E = V^2$, então mesmo as arestas são definidas parcialmente por V .

Para a entrada “in4” com um grafo não conexo, no qual apenas 1 vértice possui todos os focos, a solução é encontrada rapidamente para $V = 8$, independente do isolamento do vértice com todos os focos, ou da densidade do outro subgrafo grafo fortemente conectado, devido à estrutura do algoritmo de procurar sempre soluções menores primeiro.

Em conclusão, encontrou-se o esperado na discussão da Questão 3 quanto à complexidade do algoritmo ser altamente dependente de V para instâncias não triviais (nas quais todos os focos são encontrados por um percentual baixo de vértices conexos). Percebe-se que os outros parâmetros que definem complexidade afinal não são tão influentes como V ou são definidos por V . O número de arestas E por fim é limitado superiormente por V , e a influencia de R é meramente linear. Para a maioria das entradas, a influencia de R torna-se pouco significante, próximo à influencia do número de vértices na complexidade.

ZicaZeroZ: Uma Rede de Colaboração Coesa

Átila Martins Silva Júnior¹

¹Departamento de Ciéncia da Computaçao - Universidade Federal de Minas Gerais
(UFMG) 31.270-010 – Belo Horizonte – Brasil

amsj@dcc.ufmg.br

Resumo. *O Zica vírus é uma doença transmitida pelo mosquito Aedes aegypti que teve grande incidência no Brasil e assustou milhares de pessoas em todo mundo quando foi descoberta a relação do vírus com os casos de microcefalia. Neste trabalho, dado um grupo de voluntários e um conjunto de focos do mosquito, utilizou-se uma modelagem de grafos para formar uma rede de colaboração coesa de forma que todos os focos sejam cobertos pelos voluntários.*

1. Modelagem do Problema

Dados um conjunto de voluntários V que se relacionam através de um conjunto de laços de amizade A e um conjunto de focos do mosquito F , cada voluntário $v \in V$ tem acesso a um subconjunto de focos $F' \subset F$. O objetivo é encontrar um subconjunto $V' \subset V$ tal que todos os focos sejam acessados por pelo menos um voluntário $v \in V'$ e de forma todos os vértices $v \in V'$ se relacionam com pelo menos um dos demais vértices pertencentes a V' .

Dessa forma, para resolver esse problema foi utilizada uma modelagem de grafos. Sendo $G(V, A)$ um grafo não direcionado, onde o conjunto de voluntários são os vértices V e a relação de amizade as aresta A . Além disso, cada vértice $v \in V$ possui associado a ele um lista de focos $F' \subset F$, de forma que $F' \neq \emptyset$. O grafo G pode ser tanto conexo quanto desconexo. O algoritmo descrito a seguir mostra uma maneira de encontrar o conjunto V' dado o grafo G e o conjunto de focos F .

2. Algoritmo

O objetivo do algoritmo é encontrar o conjunto de menor número de voluntários que cobrem todos os focos e formam uma rede de amizade coesa. Isso impõe três restrições para resposta: (i) todos os focos devem ser cobertos pelos vértices e (ii) o grafo resultante deve ser conexo e (iii) o número de vértices deve ser mínimo. De forma a cumprir com esses requisitos foi proposto o algoritmo abaixo.

Data: G, F
Result: V'

```

1  $V' \leftarrow G.V$                                 // initialize
2 for  $v \in G.V$  do
3   |  $order.push(v);$ 
4 end
5 do                                              // permute the removal order
6   |  $H \leftarrow G;$ 
7   | for  $i \leftarrow 1$  to  $order.length$  do
8     |   |  $d \leftarrow DELETE-VERTEX(H,[order[i]]);$ 
9     |   | if  $CONTAINS-FOUCS(H,F) = false$  or  $CONNECTED(H) = false$ 
10    |   | then
11      |   |   |  $ADD-VERTEX(H,d);$            // get d back to H
12      |   |   |  $S.push(order[i]);$ 
13    |   | end
14  | end
15  | if  $S.length < V'.length$  then      // choose the best solution
16    |   |  $V' \leftarrow S;$ 
17 end
18 while  $PERMUTATION(order);$ 

```

Algorithm 1: Encontra o melhor conjunto de vértices que satisfaz as condições (i), (ii) e (iii).

Como se pode ver no Algoritmo 1, as entradas são o grafo G e o conjunto de focos F . Nas linhas 1 a 4 ocorre a inicialização das variáveis V' com o conjunto de vértices de G e o vetor $order$ com a ordem inicial de remoção dos vértices. Nas linhas 5 a 17 existe uma iteração que repetirá para cada ordem possível dos V vértices. Na linha 6, é realizada uma cópia de G para H , para que a configuração de G não seja perdida. Nas linhas 7 a 13, a cada iteração, será removido um vértice diferente de acordo com a ordem presente em $order$.

Após a remoção, na linha 9 é verificado se as condições (i) e (ii) são cumpridas, sendo que, a função $CONTAINS-FOUCS(H,F)$ avalia se H possui todos os focos F e a função $CONNECTED(H)$ avalia se o vértice presente em d deixa o grafo desconexo. Se alguma dessas funções foram falsas, o vértice d não pode ser removido, então ele é inserido de volta a H , linha 10, e d é adicionado à solução S , linha 11. No final da iteração, o número de vértices de H será o mínimo possível para aquela ordem de remoção. Finalmente, nas linhas 14 a 16, a cada iteração, tem-se a melhor configuração em V' e no final terá a melhor solução possível, satisfazendo a condição (iii). Nessa demonstração, considerou-se apenas o menor número de vértices para melhor configuração. Se houver empate pode-se realizar outros tratamentos (e.g., escolhendo a configuração com os vértices de menor índice).

Na próxima seção as complexidades bem como alguns detalhes adicionais da execução desse método e de todos os quais ele depende.

3. Complexidades

Primeiramente, para calcular a complexidade de tempo do Algoritmo 1 vamos analisar cada parte do código separadamente. Na inicialização de V' , linha 1, temos o custo $\Theta(V)$, pois tanto no pior quanto no melhor caso, deve-se copiar os V elementos. A inicialização do vetor $order$, linhas 2 a 4, sempre é executado V vezes, logo também será $\Theta(V)$. Logo, na inicialização, temos o custo $\Theta(V)$.

Na atribuição, linha 6, para realizar a cópia do objeto G para o H temos no pior caso o custo $O(V + F + E)$, onde $E = V^2$ e F não possui limite superior. No melhor caso, temos $\Omega(V)$, pois $F = V$ um foco para cada vértice e $E = V$ uma aresta em cada vértice.

Em seguida, linhas 7 a 13 temos uma iteração que sempre ocorre V vezes. Internamente, a função *CONTAINS-FOUCS(H,F)* verifica todos os focos de cada vértice, isso no melhor caso custo $\Omega(V)$ quando há um foco em cada vértice e o pior é $O(V + F)$. A função *CONNECTED(H)*, por sua vez, realiza um busca em profundidade em uma componente de H e verifica se todos os vértices estão conexos, no melhor caso tem custo $\Omega(V)$ quando $E = V$ e no pior caso $O(V + E)$. As funções *ADD-VERTEX(H,d)* e *S.push* têm custo 1 cada. Logo nesta iteração no melhor caso, tem custo $\Omega(V^2)$ e no pior, $O(V(V + F + E))$.

Com o objetivo e selecionar o resultado correto, as linhas 14 a 16 realizam somente uma comparação. A função *PERMUTATION(order)* da linha 17 que realiza a permutação do vetor $order$ tem custo $\Theta(V)$. Finalmente, o laço de repetição das linhas 5 à 17 repete $V!$ vezes, pois testa todas as possíveis ordens de remoção de um grafo com V vértices. O custo total da função se dá da seguinte forma: no pior caso temos o custo $O(V!(V(V + F + E)))$ e no melhor caso, $\Omega(V!(V^2))$.

O grafo foi implementado utilizando listas de adjacências. Sendo que cada vértice possui uma lista de *out links* e uma lista de focos os quais o voluntário irá atender. Sendo assim, o custo para armazenar o grafo será da ordem de $O(V + E + F)$, onde V é o número de vértices, E o número de arestas e F o número de focos.

4. Implementação

A implementação da solução para esse problema foi realizada na linguagem c++ e segue juntamente com os arquivos deste documento com nome 'ZZZ.cpp'. O programa foi compilado usando o compilador g++ versão 4.8.4 padrão c++11. Para compilar deve-se executar no terminal o shell script './compilar.sh' e para rodar os casos teste deve-se executar o comando './executar.sh <arquivo de entrada> <arquivo de saída>'.

O algoritmo funciona tanto para grafos conexos, quanto para desconexos. Sendo que se o grafo for desconexo, cada componente deve ter todos os focos. Se não houver solução o programa retornará 0.

ZicaZeroZ

Trabalho Prático 1 - Projeto e Análise de Algoritmos

Luis Fernando Miranda

¹Departamento de Ciência da Computação – UFMG – Brasil

luisfmiranda@dcc.ufmg.br

1. Introdução

Problemas envolvendo grafos são fundamentais para a área de computação. Neste trabalho, uma modelagem baseada em grafos é proposta para resolver o problema ZicaZeroZ, que visa eliminar focos de reprodução do mosquito transmissor da doença conhecida como zika.

A entrada do problema é composta por (i) um grafo onde os vértices representam um grupo de voluntários e as arestas as relações de amizade entre eles, (ii) um conjunto de focos a serem eliminados e (iii) uma relação que mostra quais focos podem ser acessados por quais voluntários. Deseja-se saber qual é o conjunto de voluntários que torna possível acessar todos os focos. O conjunto apresentado como saída deve ter o menor tamanho possível e ser composto apenas por voluntários que podem ser representados por um grafo de amizades conectado (utilizando-se as relações de amizade do grafo original).

2. Modelagem (exercício 1)

- Conjunto potência: para representar cada uma das possíveis combinações de voluntários utilizou-se uma matriz com n linhas e 2^n colunas, onde n é o número de voluntários. Cada coluna da matriz indica quais voluntários fazem parte de uma combinação específica.
- Matriz de cobertura: dada uma combinação de voluntários, a matriz de cobertura mostra quais focos estão sendo cobertos por quais voluntários.
- Vetor de focos cobertos: indica quais focos são cobertos por pelo menos um dos voluntários presentes na combinação que está sendo testada.
- Grafo de amizades: representado como uma matriz de adjacências. É utilizado apenas durante a criação do grafo induzido, uma vez que não influencia no preenchimento da matriz de cobertura.
- Grafo induzido: também implementado como uma lista de adjacências. É utilizado para verificar se a combinação de voluntários testada forma um grafo conexo, o que é feito através do algoritmo de busca em largura. Um vetor de cores é utilizado para saber se todos os vértices foram descobertos durante o algoritmo.

3. Algoritmo (exercício 2)

A solução proposta se baseia em força bruta, realizando uma busca exaustiva no espaço de soluções. A Figura 1 ilustra de forma simplificada o progresso do algoritmo durante o tratamento da entrada contida na especificação. O algoritmo começa lendo o valor de n , utilizado para determinar as dimensões da matriz de adjacências, e de m , utilizado para determinar quantas relações de amizade devem ser lidas. As m linhas seguintes são então lidas e as relações que representam são registradas na matriz de adjacências.

O algoritmo lê em seguida o valor de k , utilizado para determinar o número de focos a serem eliminados. Feito isso, as m linhas seguintes são lidas, cada uma contendo os índices dos focos aos quais cada um dos voluntários tem acesso. Será com base nesses índices que a matriz de cobertura será preenchida, a cada nova combinação de voluntários.

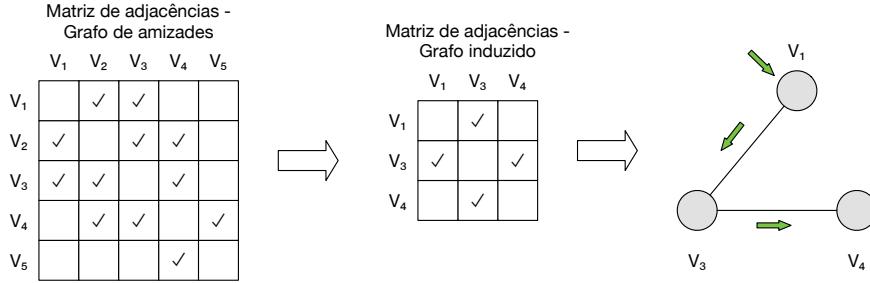
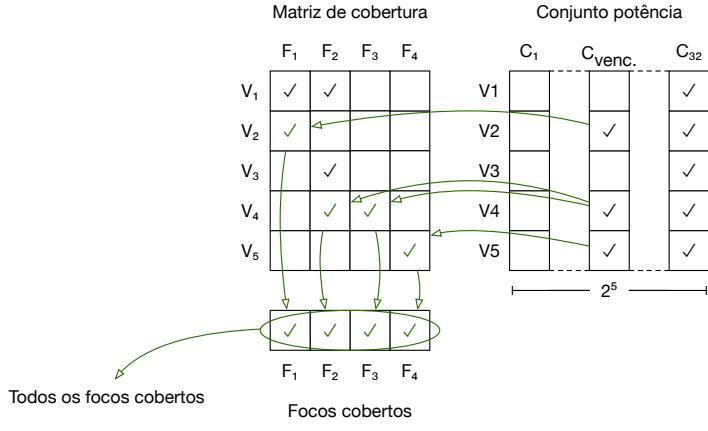


Figura 1. Progresso do algoritmo ZikaZeroZ

A próxima etapa, apresentada no algoritmo 1, envolve o teste de cada uma das possíveis combinações existentes para o conjunto \mathbb{V} de voluntários. Para que isso seja feito, criou-se uma matriz com n linhas e 2^n colunas, onde cada coluna representa uma possível combinação de voluntários (linha 1).

Para cada combinação o algoritmo executa uma série de operações para preencher o vetor que mostra quais focos foram cobertos (linhas 4-8). A ideia geral é indicar quais focos foram cobertos por pelo menos um voluntário, o que é feito através do preenchimento da matriz de cobertura, condicionado à inclusão ou não de um determinado voluntário. Em seguida, realiza-se uma série de testes buscando-se verificar (i) se o todos os focos foram cobertos (linha 10), (ii) se o grafo induzido gerado a partir da combinação de voluntários é conexo, o que é feito utilizando-se busca em largura a partir de um dos vértices que representa algum dos voluntários presentes no conjunto (linha 14) e (iii) se o número de indivíduos da combinação sendo testada é mínimo (linha 17). Se todas as três condições forem verdadeiras, o tamanho da menor solução encontrada até o momento é atualizada (linha 18) e os índices dos voluntários selecionados são armazenados na solução candidata. Em caso de empate, a solução escolhida é aquela com a menor soma dos índices dos voluntários envolvidos.

4. Análise de Complexidade (exercício 3)

4.1. Complexidade de tempo

O tempo de execução do algoritmo depende principalmente do número de voluntários, pois isso determina o número de combinações que serão testadas. Também há influência do número de ligações de amizade entre os voluntários e do número de focos a serem cobertos.

O par de loops aninhados presente na primeira parte do algoritmo (linhas 4-8) realiza $n \times k$ operações. Na segunda parte, a criação e preenchimento do grafo induzido realiza n^2 operações, enquanto o algoritmo de busca em largura tem complexidade assintótica $O(m + n)$. Cada uma dessas etapas podem ser executadas até 2^n vezes, considerando-se um caso onde todas as combinações do

Algorithm 1: ZicaZeroZ

Input: $\mathbb{V}, \mathbb{A}, \mathbb{F}, R$
Output: \mathbb{V}'

```
1 cjtoPotencia = criaCjtoPotencia( $\mathbb{V}$ .tamanho);
2 tamMenorSolucao ←  $\mathbb{V}$ .tamanho;
3 foreach combVoluntarios ∈ cjtoPotencia do
4   for i ← 1 to  $\mathbb{V}$ .tamanho do
5     if combVoluntarios[i] = true then
6       for j ← 1 to  $\mathbb{F}$ .tamanho do
7         if matrizCobertura[i][j] = true then
8           focosCobertos[j] = true;
9
10    /* neste ponto é possível saber se a combinação de
11       voluntários testada cobre todos os focos */ 
12
13    grafoAmizades = criaGrafo( $\mathbb{V}, \mathbb{A}$ );
14
15    if todas as posições do vetor de cobertura estão marcadas como true then
16      grafoInduzido = criaGrafoInduzido(grafoAmizades, combVoluntarios);
17      fonte ← primeiro elemento de combVoluntarios;
18      grafoInduzidoÉConexo ← false;
19      BFS(grafoInduzido, fonte, grafoInduzidoÉConexo);
20      if grafoInduzidoÉConexo then
21        tamCombinacaoAtual ← número de voluntários da combinação testada;
22        if tamCombinacaoAtual < tamMenorSolucao then
23          tamMenorSolucao ← tamCombinacaoAtual;
24           $\mathbb{V}'$  ← voluntários da combinação testada;
25
26
27 return  $\mathbb{V}'$ ;
```

conjunto potência formam uma solução válida. Outras etapas do algoritmo, como a criação do grafo de amizades e a verificação do tamanho das combinações, possuem custo inferior a essas operações e podem ser removidas do cálculo de complexidade. A partir disso temos que o custo assintótico para o algoritmo é dado por:

$$O(2^n \times (nk + (m + n) + n^2)) = O(2^n \times (nk + m + n^2))$$

4.2. Complexidade de espaço

O gasto de espaço do algoritmo também depende fortemente do número de voluntários, uma vez que isso determina o tamanho do conjunto potência. Há uma influência também do número de focos. O espaço necessário para armazenar as estruturas criadas tem os seguintes custos:

- Conjunto potência: $O(2^n)$
- Matriz de cobertura: $O(n \times k)$
- Vetor de focos cobertos: $O(k)$
- Grafo de amizades e grafo induzido: $O(n^2)$

Dessa forma, temos que o custo assintótico de espaço é dado por:

$$O(2^n + nk + k + n^2) = O(2^n + nk)$$

5. Implementação (exercício 4)

O trabalho foi implementado em C++ 11, em ambiente Linux, utilizando-se o compilador g++. Todos os experimentos foram executados em uma máquina com processador Inter core i7 2.9GHz, com 8GB de RAM. O padrão de entrada e saída segue o que foi estabelecido na especificação do trabalho.

6. Experimentos (exercícios 5 e 6)

A implementação do algoritmo foi testada em diversas instâncias diferentes. O objetivo foi tratar situações onde o grafo de entrada possuía características diferentes do grafo fornecido na especificação do trabalho. Um exemplo disso é o caso onde o grafo de entrada continha vértices desconectados do restante do conjunto de vértices.

Realizou-se uma análise experimental para determinar o impacto dos fatores de entrada. Como os fatores são independentes, a cada teste o valor de um fator foi alterado, enquanto os outros foram mantidos constantes. As relações de amizade foram geradas de forma aleatória, assim como os focos que cada voluntário é capaz de cobrir. As figuras seguintes ilustram os resultados obtidos:

- Impacto do número de voluntários no tempo de execução: como esperado, a variação no número de voluntários tem impacto exponencial no tempo de execução. Essa característica da curva pode ser mais facilmente verificada através do gráfico presente na Figura 3, cuja escala do eixo das ordenadas é exponencial. A instância com 25 voluntários foi a maior testada. A partir desse valor o consumo de memória fazia com que o programa fosse encerrado pelo sistema operacional.

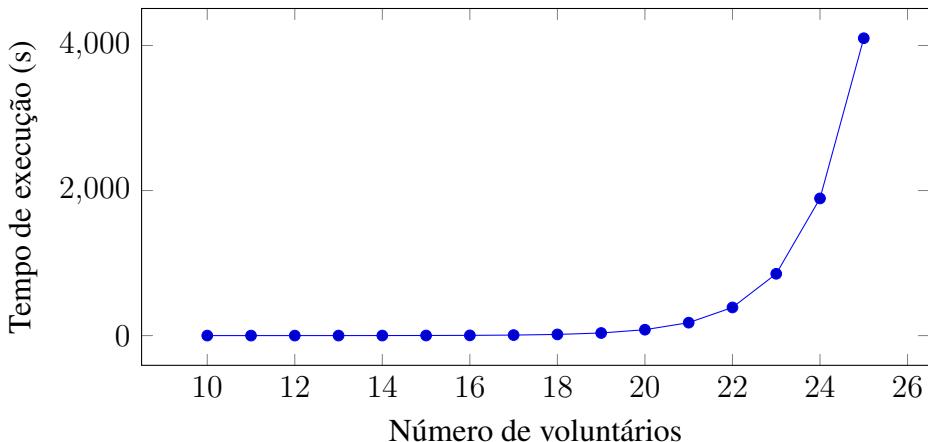


Figura 2. Tempo de execução em função do número de voluntários

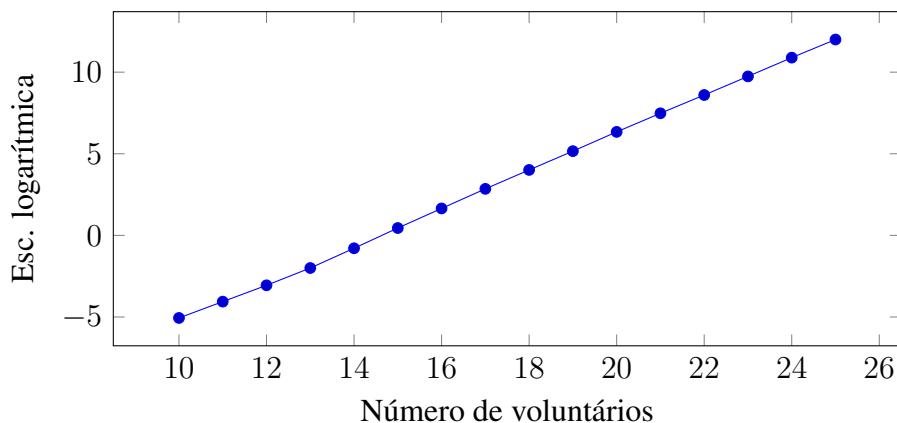


Figura 3. Tempo de execução em função do número de voluntários - Escala exponencial

- Impacto do número de voluntários na quantidade de memória alocada: o custo exponencial de espaço também foi confirmado experimentalmente e mostra que o algoritmo apresenta rendimento ruim também nesse aspecto (Figura 4). Os números do gráfico são referentes ao total de memória alocada, por isso extrapolam a quantidade de memória disponível na máquina utilizada para os testes.

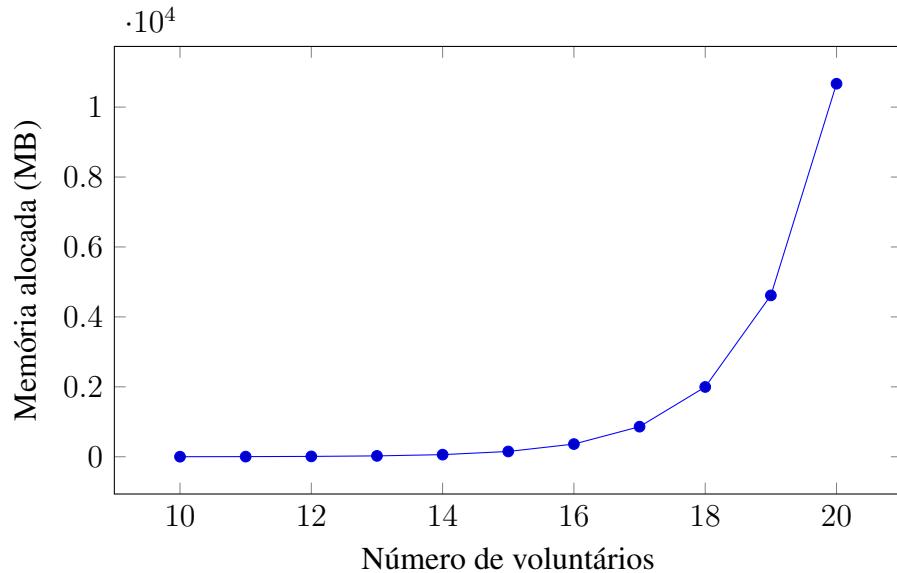


Figura 4. Memória alocada em função do número de voluntários

- Impacto do número de focos no tempo de execução: a Figura 5 mostra que a variação no número de voluntários tem impacto linear no tempo de execução, o que está de acordo com a complexidade calculada para o algoritmo.

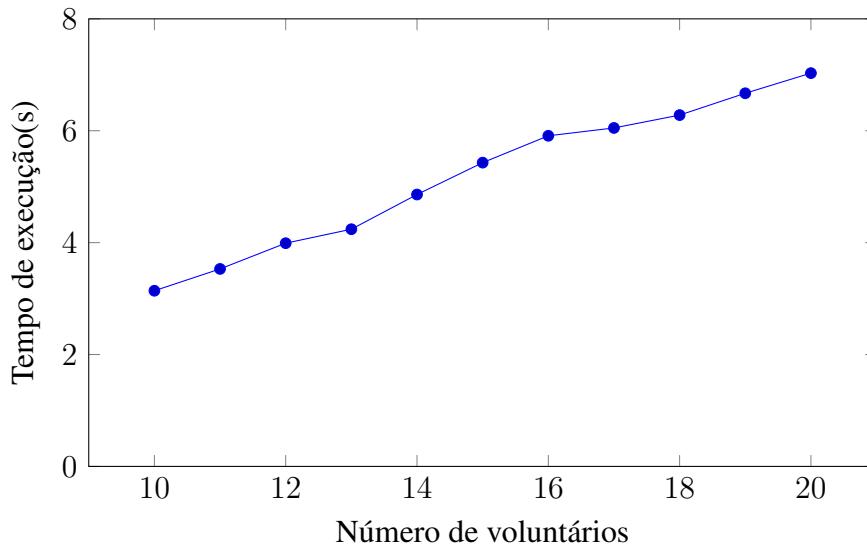


Figura 5. Tempo de execução em função do número de focos

- Impacto do número de relações de amizade no tempo de execução: ao contrário do que era esperado, o número de arestas no grafo de entrada não influenciou de forma sistemática o tempo de execução (Figura 6). Uma possível explicação para isso é que as operações que dependem das arestas do grafo de entrada só são executadas quando todos os focos são cobertos por uma determinada combinação de voluntários, o que pode não acontecer com frequência.

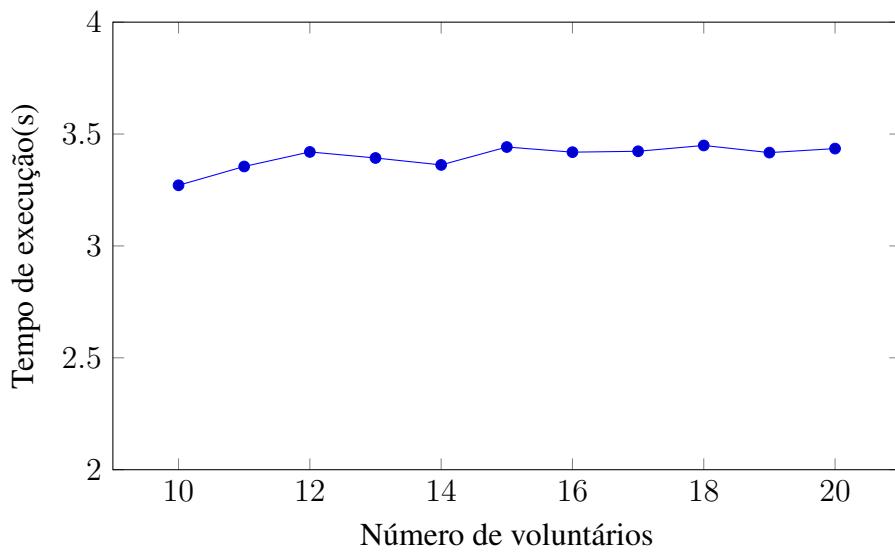


Figura 6. Tempo de execução em função do número de focos

7. Conclusão

O programa conseguiu realizar corretamente a tarefa proposta, independentemente da combinação de valores para os fatores da entrada. Entretanto, o tempo de execução e a quantidade de memória necessária para a execução do algoritmo ficaram muito acima do esperado. Ainda assim, o algoritmo sempre chegou ao final da execução com o resultado esperado, o que era o objetivo principal do trabalho.

Referências

- [1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. “*Introduction to Algorithms*”. McGraw-Hill Higher Education, 2001.

Trabalho Prático PAA

O Problema ZicaZeroZ

Michael D. Silva¹

¹ Departamento de Ciência da Computação
Universidade Federal de Minas Gerais – Belo Horizonte, MG – Brasil

{micdoug}@dcc.ufmg.br

Dados um grafo $G = (\mathbb{V}, \mathbb{A})$, em que \mathbb{V} é o conjunto de n voluntários e \mathbb{A} é o conjunto de m laços de amizade, um conjunto \mathbb{F} dos r focos de reprodução do mosquito, e, uma relação $R(v) : \mathbb{V} \rightarrow \mathbb{F}$, definida para cada $v \in \mathbb{V}$ explicitando os focos de reprodução para os quais o voluntário v tem acesso. O objetivo é selecionar o menor número de voluntários $\mathbb{V}' \subset \mathbb{V}$, tal que, todo foco é acessado, por pelo menos um voluntário $v \in \mathbb{V}'$, e o grafo induzido por \mathbb{V}' em G é conexo.

1. Apresente uma modelagem para o problema ZicaZeroZ empregando grafos. Deixe claro quais são as restrições e/ou suposições feitas.

O problema pode ser modelado utilizando dois grafos:

- O grafo de amizades $F = (\mathbb{V}, \mathbb{A})$, onde \mathbb{V} representa o conjunto dos voluntários, e \mathbb{A} as relações de amizades entre eles. Então temos um grafo onde os vértices são os voluntários e as arestas representam os laços de amizade.
- O grafo $C = (\mathbb{X}, \mathbb{R})$, onde \mathbb{X} é um conjunto formado pela união do conjunto de voluntários com o conjunto de focos que fazem parte do problema, e, \mathbb{R} representa a relação entre os voluntários e os focos aos quais eles têm acesso. Temos então um grafo bipartido, onde os vértices representam os voluntários e os focos, e as arestas ligam os voluntários aos focos que eles têm acesso.

Assim podemos aplicar uma transformação a partir do grafo F para encontrar os voluntários que devem ser selecionados, utilizando o grafo C para verificar quais focos temos acesso, validando a solução. Para que o algoritmo funcione corretamente assume-se que:

1. O grafo F é conexo, ou seja, a partir de um voluntário qualquer x existe um caminho $x \rightarrow y$ para qualquer outro voluntário y . Isso é importante pois o algoritmo proposto é derivado do algoritmo de busca em largura.
2. Para todo foco f existe pelo menos um voluntário v , tal que, a aresta (v, f) existe no grafo C . Sendo assim, todo foco é conhecido por pelo menos um voluntário. Caso contrário, não é possível encontrar uma solução para o problema.

2. Descreva um algoritmo para resolver o problema ZicaZeroZ, tendo em vista a modelagem proposta no exercício 1.

O algoritmo proposto se baseia no algoritmo clássico de busca em largura. Foi criado um procedimento denominado “BfsZica” cuja implementação é basicamente a busca em largura utilizando uma fila de prioridade, onde, os vértices retirados da fila primeiro são aqueles que adicionam mais focos desconhecidos à solução atual.

Ao fim da execução do “BfsZica” temos uma resposta válida centralizada no voluntário de onde partiu a execução da busca. O procedimento é executado a partir de todos os voluntários no procedimento “GetMinVolunteersGraph”. Após esta execução comparamos todas as respostas encontradas, retornando aquela cuja quantidade de voluntários utilizados é a menor.

No “BfsZica” quando adicionamos um voluntário que não acrescentou um novo foco ao conjunto de vértices da resposta atual, marcamos ele como ponte (“bridge”). Esta informação é utilizada no procedimento de limpeza ao final do algoritmo, onde removemos todos os vértices ponte que não tem descendentes na árvore de antecessores gerada.

A descrição do procedimento “GetMinVolunteersGraph” pode ser vista no quadro do algoritmo 1 e do “BfsZica” pode ser vista no quadro do algoritmo 2.

Algorithm 1: GetMinVolunteersGraph

Data: $F = (\mathbb{V}, \mathbb{A})$, grafo que representa voluntários e laços de amizade.
Data: $C = (\mathbb{X}, \mathbb{R})$, grafo que representa voluntários e focos conhecidos por eles.

- 1 . **Result:** $F' = (\mathbb{V}', \mathbb{A}')$, grafo que alcança todos os focos com a menor quantidade de voluntários possível.
- 2 **begin**
- 3 $vlist \leftarrow \emptyset$
- 4 **for** $v \in F.\mathbb{V}$ **do**
- 5 $vlist.append(BfsZica(v))$
- 6 **return** $\min(vlist)$

3. Analise as complexidades Temporais e Espaciais (usando notação Assintótica) do algoritmo proposto no exercício 2.

Analizando o procedimento “BfsZica” temos que a operação mais relevante é a de extrair o vértice que adiciona mais focos que é chamada na linha 11. Este método trabalha sobre a lista de vértices presentes na fronteira do grafo sendo construído durante o algoritmo.

No pior caso, cada voluntário conhece um foco distinto, e a fronteira começa com $V - 1$ vértices, onde V é o número de voluntários. A operação “max” tem complexidade temporal linear de acordo com o tamanho da lista de vértices da fronteira (neste caso começa com $V - 2$ comparações). Sabendo disso podemos definir no pior caso que a complexidade do procedimento “BfsZica” é $f(n) = \sum_{i=1}^{V-2} i \rightarrow \frac{(V-1)(V-2)}{2} \rightarrow \frac{V^2-3V+2}{2}$.

Como o procedimento “BfsZica” é chamado para todos os vértices do grafo F , ou seja, V vezes, temos que a complexidade temporal do algoritmo para o pior caso é $f(n) = V \frac{V^2-3V+2}{2} \rightarrow O(V^3)$.

Em relação à complexidade de espaço, o algoritmo deve ser capaz de representar os dois grafos citados no exercício 1 utilizando a implementação por lista de adjacência, logo temos que a complexidade é de $O((V + A) + (X + R))$, onde V é o número de voluntários, A a quantidade de amizades, X a quantidade de voluntários e focos, e, R a quantidade de vínculos entre voluntários e focos.

Algorithm 2: BfsZica

Data: $F = (\mathbb{V}, \mathbb{A})$, grafo que representa voluntários e laços de amizade.
Data: $C = (\mathbb{X}, \mathbb{R})$, grafo que representa voluntários e focos conhecidos por eles.
Data: $orig$, vértice origem da busca.
Data: $totalFocuses$, lista de todos os focos.
Result: $F' = (\mathbb{V}', \mathbb{A}')$, grafo induzido em \mathbb{F} a partir de $orig$ que é conexo e alcança todos os focos.

```
1 begin
2     Inicializa todo  $v \in F.\mathbb{V}$  com
3          $v.d = \infty, v.visited = false, v.pi = null, v.bridge = false$ 
4          $orig.visited \leftarrow true$ 
5          $orig.d \leftarrow 0$ 
6          $frontier \leftarrow list(F.adj(orig))$ 
7         foreach  $v \in F.adj(orig)$  do
8              $v.pi \leftarrow orig$ 
9              $v.d \leftarrow 1$ 
10         $focuses \leftarrow C.adj(orig)$ 
11        while  $focuses.length != totalFocuses.length$  do
12             $u \leftarrow max(frontier)$ 
13             $u.visited \leftarrow true$ 
14             $new\_focuses \leftarrow focuses - C.adj(u)$ 
15            if  $new\_focuses.length == 0$  then
16                 $u.bridge \leftarrow true$ 
17                 $focuses \leftarrow focuses \cup new\_focuses$ 
18                foreach  $v \in F.adj(u)$  do
19                    if  $v.d == \infty$  then
20                         $frontier.append(v)$ 
21                         $v.d \leftarrow u.d + 1$ 
22                         $v.pi \leftarrow u$ 
23
24    return criaGrafoSimplificado(friendships.\mathbb{V})
```

4. Implemente o algoritmo proposto no exercício 2 na linguagem de programação C, C++, JAVA ou PYTHON.

A implementação em PYTHON segue em anexo ao relatório.

5. Execute testes da implementação do Exercício 4, propondo instâncias interessantes para o problema ZicaZeroZ. Uma sugestão é que seja produzido um gráfico do tempo de execução por tamanho de entrada (n , m , r). Outra sugestão é relatar o tamanho da maior instância para a qual foi produzida uma solução.

Foram executados testes com os exemplos disponibilizados no fórum da disciplina. Os seguintes resultados foram encontrados:

Arquivo	n	m	r	Comparações
in0	5	6	4	16
in1	6	6	6	17
in2	7	6	13	69
in3	8	28	8	168
in4	4	3	4	3
in5	8	21	8	105
in6	5	4	5	3
in7	5	10	8	25

Tabela 1. Tabela com testes a partir dos exemplos postados no forum da disciplina

Também foram executados testes para instâncias do pior caso relatado no exercício 3. Foram utilizados como entrada grafos completos com a quantidade de vértices variando de 50 a 800. Os resultados podem ser vistos no gráfico 1. A maior instância testada foi de 800 vértices. Grafos maiores que este levam mais de 9 minutos para serem analisados.

6. Compare a análise e a execução respectivamente, exercícios 3 e 5.

Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

Ao comparar os resultados dos testes com a previsão teórica vemos que a equação para o pior caso é refletida no exemplo do arquivo “in3” que contém exatamente o cenário citado no exercício 3. Para os casos em que temos um grafo esparsa, vemos que a quantidade de comparações é extremamente menor, como pode ser visto no exemplo do arquivo “in0”, “in1”, “in4” e “in6” na tabela 1. O tempo de execução não foi levado em consideração nesta tabela, pois para todos os exemplos não foi alcançado 1 segundo completo.

Para verificar a evolução do tempo de execução do algoritmo para entradas maiores foram feitos testes de execução usando grafos completos onde cada vértice só conhece um foco, variando de 50 a 800 vértices incrementando em 50. Como pode ser visto no gráfico 1, a evolução do tempo de execução do algoritmo também se mostra condizente com a análise teórica proposta.

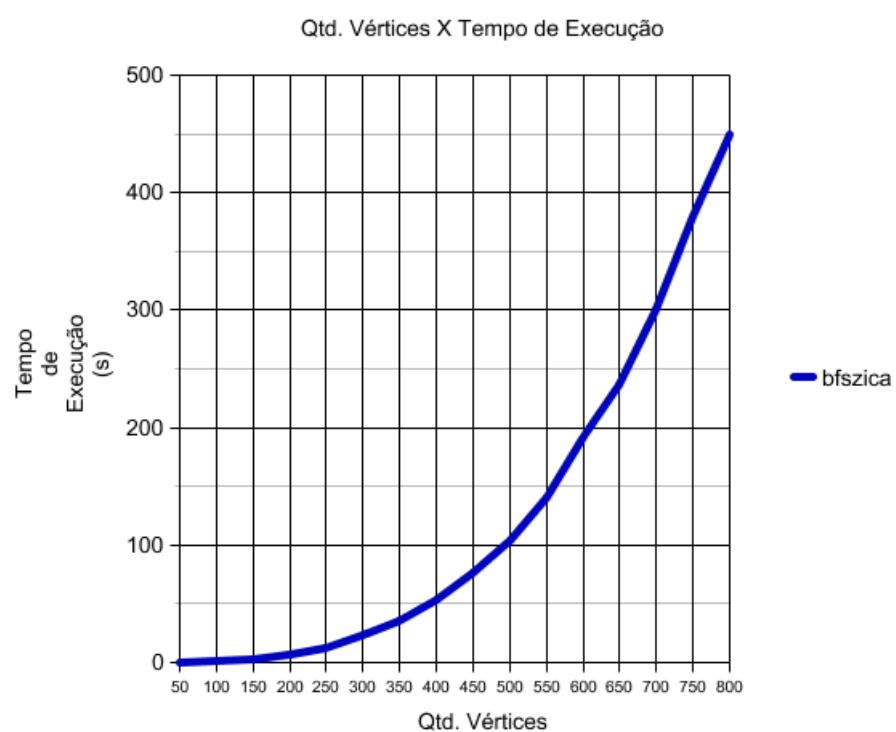


Figura 1. Tempo de execução do algoritmo para entradas com grafos completos variando de 50 a 800 vértices

Trabalho Prático: ZikaZeroZ

José Laerte Xavier Pires Xavier Júnior¹

¹Departamento de Ciência da Computação (DCC) – UFMG
Av. Pres. Antônio Carlos, 6627 – Pampulha, Belo Horizonte – MG, 31270-901

laertexavier@dcc.ufmg.br

Resumo. Este documento tem como objetivo descrever e analisar os resultados obtidos com a resolução do problema **ZikaZeroZ**. O problema consiste em determinar, dentre os diversos voluntários da Geração Z, uma rede de colaboração coesa que permita combater todos os focos do mosquito transmissor (*Aedes aegypti*). A fim de alcançar uma solução ótima, foi utilizado um algoritmo exaustivo, que verifica dentre os subconjuntos de voluntários dados, aqueles capazes de combater todos os focos da doença. Dessa forma, este relatório apresenta a solução e analisa-a em termos de tempo de execução e utilização de memória, comparando os valores teóricos e obtidos em execuções experimentais. A estrutura do mesmo divide-se, portanto, em conformidade com a especificação do trabalho, sendo cada Seção referente a uma questão levantada pelo mesmo.

1. Apresentação

O vírus da zika, transmitido em humanos pelo mosquito *Aedes aegypti*, causa a doença mundialmente conhecida como zika. O seu combate acontece por meio da eliminação de focos de reprodução do mosquito transmissor, a fim de evitar que o vírus se propague. Para tanto, o problema **ZikaZeroZ** utiliza o fenômeno mundial das redes sociais virtuais para criar uma rede de colaboração coesa de voluntários da Geração Z. Esses voluntários tem acesso a focos específicos da doença e podem, portanto, combatê-los de maneira direta.

O problema, então, consiste em: dado um conjunto V de voluntários associados mutualmente pela possível relação de amizade entre eles e um conjunto F de focos, ambos relacionados pelo critério de acessibilidade, espera-se encontrar o **menor** subconjunto V' de voluntários v tal que (i) exista uma relação de amizade entre todos (ou seja, não exista nenhum voluntário isolado) e (ii) todos os focos em F sejam acessíveis (ou seja, combatíveis).

Dessa maneira, o problema pode ser modelado utilizando grafos: dados os conjuntos V de n voluntários e A de m laços de amizade entre eles, podemos construir um grafo G , tal que $G = (V, A)$. Assim, cada voluntário representa um vértice do grafo e os laços de amizade entre eles representam as arestas. Além disso, cada voluntário v está associado a um conjunto de focos F os quais eles podem combater. Essa relação pode ser modelada associando a cada vértice o subconjunto F' de focos acessíveis pelo voluntário v .

O objetivo, então, é encontrar o **menor** subgrafo $G' = (V', A')$ tal que (i) todos os voluntários $v \in V'$ tenham entre eles uma relação de amizade em A' (ou seja, o grafo G'

seja conexo); e (ii) todos os focos $f \in F$ sejam acessados pela união dos voluntários $v \in V'$.

Para tanto, algumas suposições precisam ser apresentadas:

- O grafo G é não-direcionado e não-valorado (não possui valor nas arestas);
- Não existe nenhuma relação de amizade de um voluntário com ele mesmo (ou seja, não existe em G nenhum *self-loop*);
- Podem existir mais de um subgrafo G' que satisfaçam as condições (i) e (ii) (nesse caso, usaremos como critério de desempate a menor soma dos ID' s dos vértices).

2. Descrição do Algoritmo

A fim de gerar uma solução para o problema **ZikaZeroZ** descrito em 1, podemos utilizar um algoritmo de busca exaustiva em todos os possíveis subconjuntos de V , verificando as condições (i) e (ii). Os subconjuntos de menor cardinalidade que satisfazem esses critérios são, então, elegíveis como resultado. Aquele que satisfizer o critério de desempate (menor soma dos ID' s) é retornado como resposta. O Algoritmo 1 descreve essa solução.

Algorithm 1: GetCollaborationNetwork

```

input : Grafo  $G = (V, A)$ , conjunto  $F$  de focos.
output: Conjunto resposta  $V' \subset V$ .

1  $P \leftarrow PowerSet(V)$ 
2  $PossibleAnswers \leftarrow \emptyset$ 
3  $smallerAnswerSize \leftarrow \infty$ 

4 foreach  $V' \subset P$  do
5   if  $|V'| > smallerAnswerSize$  then
6     continue
7   end
8   if  $reachedFocus(V') == |F|$  and  $isConnected(G, V')$  then
9     if  $|V'| == smallerAnswerSize$  then
10       $PossibleAnswers \leftarrow PossibleAnswers \cup V'$ 
11    end
12    if  $|V'| < smallerAnswerSize$  then
13       $PossibleAnswers \leftarrow V'$ 
14       $smallerAnswerSize \leftarrow |V'|$ 
15    end
16  end
17 end

18 return  $chooseAnswer(PossibleAnswers)$ 

```

Como podemos observar, o algoritmo inicia gerando o *conjunto das partes* para o conjunto de vértices V (Linha 1). A ideia principal é iterar sobre esse conjunto (Linhas 4-17), verificando se o grafo induzido por $V' \subset P$ satifaz as condições (i) e (ii). Como o problema requer o conjunto V' com menor cardinalidade, verificamos também o tamanho dos conjuntos elegíveis V' (Linhas 9 e 12) e finalizamos aplicando o critério de desempate (Linha 18).

Uma vez que o algoritmo percorre todos os possíveis subconjuntos de V gerados pela função `PowerSet` (V) na Linha 1, podemos considerar que o algoritmo, embora não muito eficiente, sempre nos retornará uma solução ótima. Além disso, através do uso das duas funções auxiliares `reachedFocus` (V') e `isConnected` (G, V') (Linha 8), podemos garantir que as condições impostas serão sempre avaliadas e que todos os subconjuntos $V' \subset P$ em `PossibleAnswers` as satisfazem.

Por fim, observamos que a variável `smallerAnswerSize` armazenará sempre a cardinalidade dos subconjuntos $V' \subset P$ em `PossibleAnswers`. Ela se torna útil para evitarmos verificar e processar subconjuntos V' cuja cardinalidade seja maior que aquelas possíveis respostas já encontradas, contribuindo para amenizar o tempo de processamento da solução.

3. Análise de Complexidade

Na Seção 2 apresentamos um algoritmo para resolver o problema do **ZikaZeroZ** estudado nesse trabalho. Ele baseia-se na busca exaustiva por um subconjunto $V' \subset P(V)$ que satisfaça as condições impostas pelo problema. Por se tratar de um algoritmo força bruta (dada a exigência de uma solução ótima), sabemos que a sua complexidade é, no mínimo, exponencial. As Subseções 3.1 e 3.2 detalham as análises temporal e espacial, respectivamente.

3.1. Análise Temporal

Para fazermos a análise temporal do Algoritmo 1, vamos, inicialmente, analisar assintoticamente cada uma das operações mais custosas:

1. `PowerSet` (V): Esta operação constrói, a partir do conjunto V , o seu respectivo conjunto das partes. Ela pode ser modelada também como um grafo $P = (V', E')$, onde cada vértice $v' \in V'$ representa um subconjunto desse conjunto potência e as arestas representam as ligações entre os estados. Para obter o conjunto $P(v)$, podemos fazer uma busca em largura (BFS) nessa grafo, de custo $O(V' + E')$. Como as cardinalidades dos conjuntos V' e E' são, proporcionalmente, iguais a 2^V , podemos concluir que o custo total dessa função é, no pior caso, $\mathbf{O}(2^V)$.
2. `reachedFocus` (V'): A operação de focos alcançáveis soma a quantidade de focos que os vértices do subconjunto V' alcançam. Para tanto, ele percorre cada uma desses vértices, acessando o número de focos alcançáveis pelos mesmos e somando a uma variável total. No pior caso, o subconjunto V' terá cardinalidade igual ao conjunto V . Dessa forma, essa operação terá custo de $\mathbf{O}(V)$.
3. `isConnected` (G, V'): Para avaliar se um subconjunto de vértices V' é conexo, iteramos sobre cada um dos seus vértices u' , verificando se existe alguma aresta no grafo original G que o conecte com todos os outros vértices v' . Ou seja, avaliamos se, para todo $u' \in V'$ e $v' \in V'$, existe uma aresta $(u', v') \in G.A$. Como a cardinalidade do conjunto V' , no pior caso, é igual a V , podemos afirmar que a complexidade dessa função é $\mathbf{O}(V^2)$.

- chooseAnswer (PossibleAnswers): Esta operação escolhe, dentre todas as soluções elegíveis, aquela que possui a menor soma dos seus ID's. Para tanto, ela percorre todos os elementos da lista PossibleAnswers e, para cada um, percorrer seus $v' \in V'$ elementos, somando seus ID's. Considerando que, no pior caso, cada uma das k possíveis soluções terá cardinalidade V , o custo dessa operação é $O(kV)$, com k representando o tamanho da lista PossibleAnswers de possíveis soluções.

Para descobrir o custo total do Algoritmo 1, vamos somar todos os custos descritos, desconsiderando o custo das demais operações, uma vez que são constantes. Dessa forma, somamos o custo de uma operação PowerSet (V) (Linha 1), mais o custo das operações reachedFocus (V') e isConnected (G, V') executados 2^v vezes (Linhas 4-17), mais o custo da operação chooseAnswer (PossibleAnswers). O resultado é, portanto:

$$O(2^V + 2^V(V + V^2) + kV)$$

Como estamos interessado na complexidade do Algoritmo utilizando notação assintótica, vamos considerar apenas o termo de maior custo do somatório. Portanto, podemos concluir que o custo total é:

$$O(2^V(V + V^2))$$

3.2. Análise Espacial

O Algoritmo 1 possui apenas duas operações que requerem um custo espacial maior. Analisaremos cada uma delas para, por fim, computar a complexidade espacial do mesmo:

- PowerSet (V): Esta operação gera um conjunto de 2^V subconjuntos que representa o conjunto das partes $P(v)$. Cada subconjunto possui, no máximo, V elementos. Dessa forma, a sua complexidade espacial é $O(V2^V)$.
- chooseAnswer (PossibleAnswers): Após iterar sobre todas as possíveis soluções, a lista PossibleAnswers armazena todas as soluções elegíveis para o problema. Considerando que, no pior caso, todos os 2^V subconjuntos sejam respostas possíveis, a complexidade espacial desta operação é $O(2^V)$.

Para todas as outras operações, um custo constante é verificado. Dessa forma, temos que a complexidade espacial do Algoritmo é a soma da complexidade das duas operações anteriores:

$$O(V2^V + 2^V)$$

Descartando o termo de menor complexidade, concluímos que a complexidade espacial é:

$$O(V2^V)$$

4. Implementação

A fim de testar o Algoritmo 1 com entradas interessantes e fazer as medições experimentais descritas na Seção 5, fizemos uma implementação do mesmo utilizando a linguagem

```

public ArrayList<Integer> getCollaborationNetwork(){
    Set<Set<Vertex>> powerSet = Utils.powerSet(volunteerGraph.getSetOfVertices());

    List<Set<Vertex>> possibleAnswer = new ArrayList<Set<Vertex>>();
    int smallestPossibleAnswer = Integer.MAX_VALUE;

    for (Set<Vertex> subgraph : powerSet) {
        if(subgraph.size() > smallestPossibleAnswer)
            continue;

        if ((this.reachedFocus(subgraph) == this.numberOfFocus)
            && (Utils.isConnected(volunteerGraph, subgraph))) {
            if(subgraph.size() == smallestPossibleAnswer){
                possibleAnswer.add(subgraph);
            }

            if(subgraph.size() < smallestPossibleAnswer){
                possibleAnswer.clear();
                possibleAnswer.add(subgraph);
                smallestPossibleAnswer = subgraph.size();
            }
        }
    }

    return this.chooseAnswer(possibleAnswer);
}

```

Figura 1. Implementação em Java do Algoritmo 1

Java. A Figura 1 apresenta o método `getCollaborationNetwork()` correspondente.

Por se tratar de uma linguagem Orientada a Objeto, utilizamos algumas Classes para representar estruturas de grafos e resolver a instância do problema. Essas estruturas serviram de arcabouço para o algoritmo e estão descritas brevemente a seguir:

- **ZikaZeroZ**: Classe que resolve a instância do problema **ZikaZeroZ**, aplicando o Algoritmo 1 e seus métodos auxiliares.
- **Graph**: Objeto que representa um Grafo $G = (V, E)$ através de uma lista de adjacências. Operações sobre este grafo estão implementadas nesta classe.
- **Vertex**: Objeto que representa cada um dos vértices do grafo, com uma lista dos focos acessíveis pelo mesmo.
- **Utils**: Classe de operações gerais que implementa algumas funções auxiliares tais como `isConnected(G, V')` e `powerSet(V)`.
- **Main**: Classe geral que recebe os dados de entrada em arquivo, no formato especificado no problema, monta o grafo correspondente, gera a solução para essa instância do **ZikaZeroZ** e imprime resultado no arquivo de saída.

Todas as classes e objetos do sistema foram desenvolvidos com o objetivo de desacoplar o sistema ao máximo e utilizar suas operações e objetos em outras instâncias de problemas modelados com grafo.

5. Execução Experimental

5.1. Caracterização dos Experimentos

Os experimentos descritos nas Subseções 5.2 e 5.3 foram realizados em um notebook Dell Inspiron 5448 com as seguintes configurações:

- **Processador:** Intel(R) Core(TM) i5-5200U CPU @ 2.20 GHz
- **Memória RAM:** 8 GB
- **HDD:** 1 TB
- **Sistema Operacional:** Windows 10

Para coletar os valores de tempo de execução, o código foi instrumentalizado utilizando o método `System.currentTimeMillis()` de Java. Já para os valores de uso de memória, utilizamos os métodos `Runtime.totalMemory()` e `Runtime.freeMemory()`.

É importante, ainda, destacarmos que os valores obtidos são referentes ao cálculo da solução do problema (ou seja, do método `getCollaborationNetwork()`). As operações de leitura e escrita de arquivo, bem como a montagem do grafo, não foram consideradas, uma vez que são secundárias ao algoritmo em estudo.

5.2. Testes de Implementação

Para avaliarmos o Algoritmo 1 executamos a implementação desenvolvida e apresentada na Seção 4 com uma bateria de instâncias, entre as quais encontram-se as disponibilizadas com a especificação do problema. A Tabela 1 apresenta um resumo dos tamanhos das entradas (m , n e r) e os respectivos resultados para tempo de execução (em milissegundos) e uso de memória (em kb).

Instância	n	m	r	Tempo	Memória
in0	5	6	4	4	1331
in1	6	6	6	4	1331
in2	7	6	13	4	1331
in3	8	28	8	6	1996
in4	4	3	4	2	1331
in5	8	21	8	9	1997
in6	5	4	5	2	1331
in7	5	10	8	3	1331
in8	8	7	8	5	1996
in9	5	5	6	2	1331
in10	4	3	7	3	1331
in11	12	66	12	31	5990
in12	1	0	4	2	1331
in13	8	21	8	4	1997
in14	5	7	4	3	1331
in15	3	2	3	2	1331

Tabela 1. Caracterização dos Experimentos

Dentre as instâncias testadas, algumas apresentam configurações críticas, com condições que testam o problema de várias maneiras diferentes. A instância `in0`, por exemplo, testa o algoritmo com duas possíveis soluções de tamanho mínimo, porém uma conexa ($\{2, 4, 5\}$) e outra não ($\{1, 4, 5\}$). Já a instância `in14` possui as mesmas possíveis soluções, mas, desta vez, ambas conexas. A Figura 2 apresentam ambas as instâncias em (a) e (b), respectivamente.

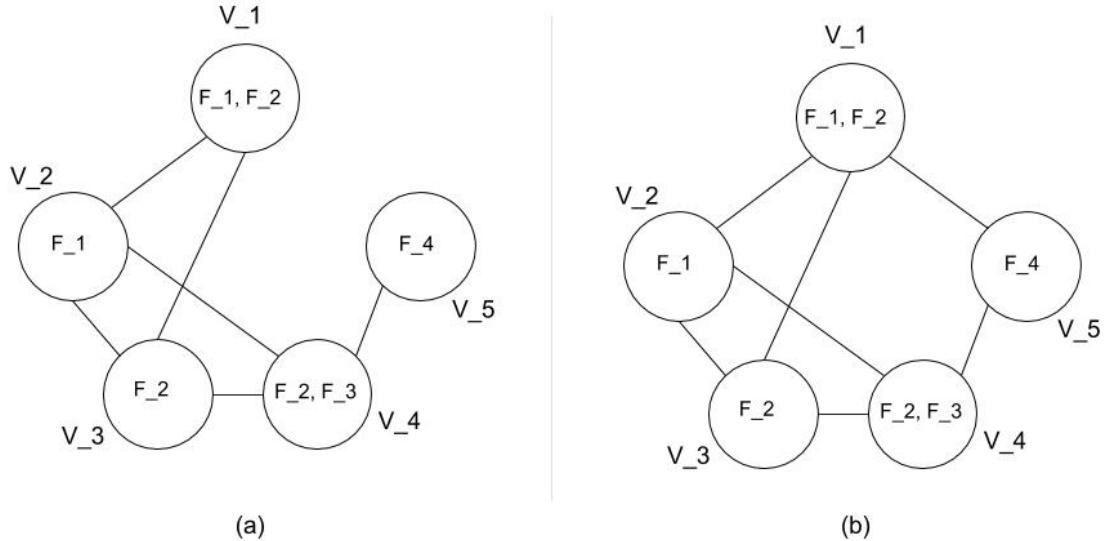


Figura 2. Teste de comparação de possíveis soluções de mesmo tamanho: $\{1, 4, 5\}$ e $\{2, 4, 5\}$ para ambos. Em (a) o conjunto solução é o $\{2, 4, 5\}$ já que o conjunto $\{1, 4, 5\}$ não é conexo. Já em (b) o conjunto $\{1, 4, 5\}$ é escolhido pelo critério de desempate.

Outra configuração interessante testada é encontrada em `in15`, representada na Figura 3. Como podemos observar, duas soluções são possíveis: $\{1, 2\}$ e $\{3\}$. O algoritmo deve, portanto, escolher a solução de menor cardinalidade ($\{3\}$), respeitando a exigência de encontrarmos o **menor** conjunto de voluntário V' .

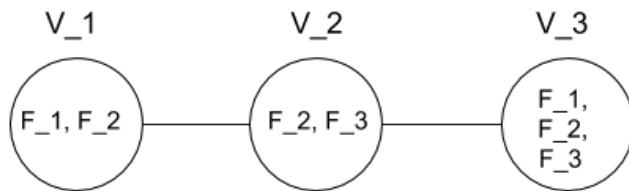


Figura 3. Teste de comparação de possíveis soluções de tamanhos diferentes: $\{1, 2\}$ e $\{3\}$. A solução $\{3\}$ é escolhida.

Além dessas, outras 13 instâncias foram testadas, exercitando aspectos importantes das entradas e dos algoritmos. Para todas elas, obtivemos os resultados esperados para

o problema¹. A Tabela 1 apresenta, por fim, as características de cada uma das entradas, bem como os resultados de medição temporal e espacial. Esses valores serão importantes para a análise na Subseção 5.3 e na Seção 6.

5.3. Testes de Desempenho

A análise do algoritmo na Seção 3 mostra que a sua complexidade é proporcional ao número de vértices do grafo. Dessa forma, a fim de testamos o desempenho temporal e espacial da nossa implementação, executamos uma sequência de testes com instâncias variando o número n de voluntários e mantendo fixos os valores m e r em 0 e 5, respectivamente.

A Tabela 2 apresenta os resultados obtidos para cada uma das instâncias, com uma variação de 5 voluntários por instância. O tempo é dado em milissegundo e a taxa de utilização de memória em kb.

Instância	n	m	r	Tempo	Memória
de01	1	0	5	1	1331
de02	5	0	5	3	1331
de03	10	0	5	16	1331
de04	15	0	5	145	23616
de05	20	0	5	5724	586576
de06	25	0	5	-	-
de06	24	0	5	-	-
de06	23	0	5	-	-
de07	22	0	5	-	-
de08	21	0	5	17568	1202923

Tabela 2. Testes de Desempenho: variação de n , com $m = 0$ e $r = 5$.

Como podemos observar, o tempo de execução e o uso de memória cresce de maneira diretamente proporcional ao número de voluntários. Além disso, podemos verificar que no teste da variação de 20 para 25 voluntários, o sistema estourou a memória e não gerou resultado algum. Por isso, executamos testes variando o valor de n decrementando em 1 unidade por instância, a fim de descobrir o maior valor possível de execução. Encontramos, dessa forma, que **a maior instância para a qual foi produzida uma solução é com 21 voluntários**.

Sabendo do resultado anterior, executamos novamente a implementação com o valor de n variando de 1 até 21 e geramos os gráficos da Figura 4 com os valores de tempo de execução e utilização de memória, dados em milissegundos e kb, respectivamente.

6. Discussão

Com os resultados obtidos nas Seções 3 e 5.3, podemos fazer uma última análise acerca das previsões teóricas e os resultados práticos obtidos nas execuções. Para tanto, observa-

¹ As entradas e saídas testadas estão disponíveis no pacote `testes` da solução.

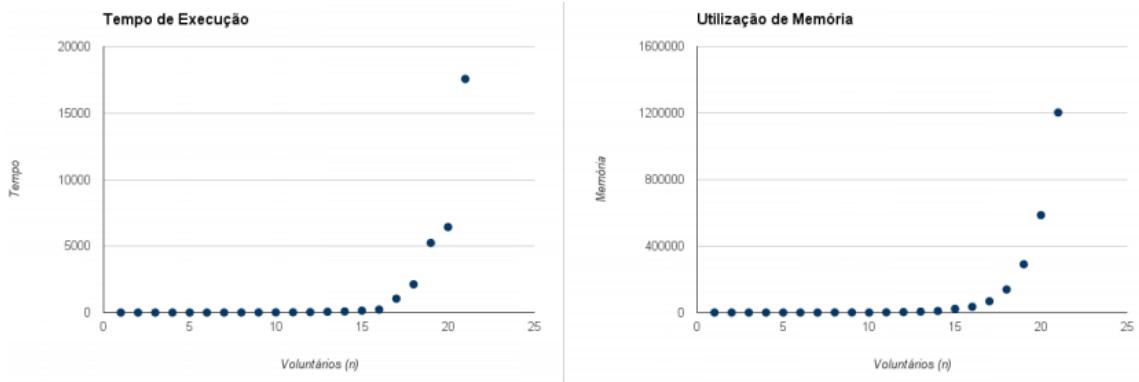


Figura 4. Tempo de Execução e Utilização de Memória para execuções de n variando de 1 a 21.

mos o comportamento da curva gerada na Figura 4 para os valores de tempo de execução e utilização de memória.

Como o algoritmo analisado é da classe de complexidade exponencial, uma vez que é força bruta, verificamos que os valores obtidos nas execuções estão em conformidade com essa ordem de grandeza, mostrando que as previsões teóricas estão de acordo com os valores reais. Os gráficos da Figura 5 apresentam os valores resultantes das execuções da Tabela 2, comparados a função exponencial $f(n) = 2^n$.

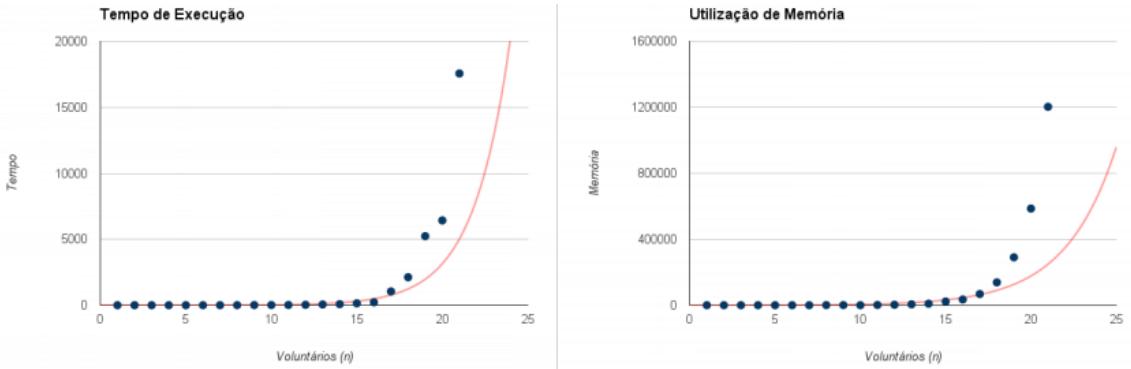


Figura 5. Comparaçāo dos valores obtidos com uma função exponencial $f(n) = 2^n$.

Como podemos observar, os valores obtidos são da mesma ordem de grandeza e ambas as curvas (temporal e espacial) possuem o mesmo formato da função teórica. Entretanto, a função de complexidade descrita na Seção 3 é apenas *proporcional* a função exponencial $f(n) = 2^n$, justificando o fato da curva dos valores obtidos estarem "deslocadas" nos gráficos apresentados.

Para finalizar a nossa análise, calculamos os valores obtidos com as funções de complexidade descritas na Seção 3 com os valores de n utilizados na Tabela 2, com o objetivo de verificarmos numericamente os valores teóricos e os resultados obtidos. A Tabela 3 detalha os resultados.

n	Tempo de Execução		Utilização de Memória	
	Teórico	Prático	Téorico	Prático
1	4	1	2	1331
5	960	3	160	1331
10	112640	16	10240	1331
15	7864320	145	491520	23616
20	440401920	5724	20971520	586576
21	968884224	17568	44040192	1202923

Tabela 3. Comparação dos valores teóricos e reais obtidos nas Seções 3 e 5.3.

A partir da comparação dos valores teóricos e práticos apresentados anteriormente, observamos que os resultados práticos mostram valores consideravelmente menores que os teóricos. Isto pode ser explicado pelo fato de a análise teórica da Seção 3 considerar o pior caso do algoritmo em estudo. Entretanto, sabemos que, na prática, o pior caso não ocorre com frequência e, por isso, nem sempre serão obtidos nos testes experimentais.

7. Conclusão

Neste trabalho, analisamos o problema do **ZikaZeroZ**, que consiste em determinar, dentre uma rede de voluntários da Geração Z, uma rede de colaboração coesa que cubra todos os focos da doença zika com o menor número de voluntários interconectados por laços de amizade. Para tanto, o problema foi modelado utilizando grafos e, então, foi descrito e analisado um algoritmo de busca exaustiva que o solucione. As análises experimentais foram coletadas a partir de uma implementação real do problema utilizando a linguagem Java e os valores obtidos foram comparados com as avaliações teóricas desenvolvidas.

Podemos concluir, dessa maneira, a ineficiência prática dos algoritmos do tipo força bruta, uma vez que, neste caso, ele nos fornece solução apenas para um conjunto de no máximo 21 voluntários. Além disso, observamos a discrepância que normalmente obtemos entre os valores de tempo de execução e utilização de memória obtidos em execuções experimentais e análises teóricas de pior caso. Verificamos, por fim, que apesar dessa discrepância, os valores se comportam assintoticamente de maneira semelhante e, portanto, conservam a ordem de grandeza da classe a qual pertencem.

Trabalho Prático de Grafos

Gabriel de Biasi¹

¹ Departamento de Ciência da Computação

Universidade Federal de Minas Gerais

Av. Antônio Carlos, 6627 – Pampulha – Belo Horizonte – MG

biasi@dcc.ufmg.br

1. Modelagem do Problema

O problema ZicaZeroZ foi modelado utilizando um grafo não-orientado, onde os vértices do grafo representam os voluntários, as arestas representam as relações de amizades entre os voluntários e os focos de zica que os voluntários tem acesso são armazenados como um meta-dado dentro do vértice.

A proposta é projetar um algoritmo de programação dinâmica, onde sejam calculados os conjuntos de vértices que visitam no máximo um foco de zica, depois os conjunto de vértices que visitam no máximo dois focos de zica utilizando as soluções anteriores já encontradas, depois para três focos e assim por diante, até que o número de focos visitados seja igual ao valor de r .

Nesta estratégia, cada elemento da matriz de soluções $F(v, p)$ representam uma lista de subgrafos conexos $\mathbb{V}' \subset \mathbb{V}$ de no máximo $v + 1$ vértices, que visitam exatamente $p + 1$ focos de zica.

Esta tipo de aproximação se torna muito efetiva para grafos esparsos e que tenham uma solução ótima muito pequena. Para grafos densos ou de solução $\mathbb{V}' \approx \mathbb{V}$, o custo pode ser exponencial.

2. O algoritmo

A partir da modelagem do problema, a resolução possui cinco algoritmos distintos, sendo eles *Get_Places_Amount*, *Get_Smaller_Sum*, *Combine_Solution* e o *Execute*.

Nos algoritmos abaixo, as listas possuem um atributo especial, chamado *length*, que retorna o tamanho de uma lista. Há também um método chamado *add(p)* que adiciona o elemento p no final de uma lista. A operação de união (\cup) cria uma nova lista sem elementos repetidos. O conjunto \mathbb{V} representa o conjunto de vértices do grafo do problema e o conjunto \mathbb{R} representa o conjunto de focos de zica do problema.

O algoritmo *Get_Smaller_Sum* é utilizado apenas no final para critério de desempate. O algoritmo recebe por parâmetro uma lista de subgrafos conexos que são soluções ótimas do problema e retorna o subgrafo que contém a menor soma dos identificadores de seus vértices. Pela sua trivialidade, não é necessário explicar o algoritmo com mais detalhes.

O algoritmo *Get_Places_Amount* recebe um subgrafo conexo $\mathbb{V}' \subset \mathbb{V}$ e calcula o número de focos de zica que são visitados por ele, através da união dos conjuntos de focos entre todos os vértices do subgrafo.

Algorithm 1: Obtem a quantidade de focos visitados pelo subgrafo G'

```

1 Function Get_Places_Amount( $G$ ) :
2    $all\_places \leftarrow \{\}$ 
3   foreach  $v \in G$  do
4      $all\_places \leftarrow all\_places \cup v.places$ 
5   return  $all\_places.length$ 

```

O algoritmo *Combine_Solution* recebe dois subgrafos conexos $\mathbb{V}', \mathbb{V}'' \subset \mathbb{V}$ e tenta combinar os dois subgrafos em apenas um subgrafo conexo, esperando um tamanho exato de $i + 1$. Se não houver pelo menos uma aresta entre os vértices dos subgrafos para torná-lo conexo, o algoritmo encerra a execução. Se o subgrafo resultante for válido, ele é colocado na posição da matriz F de acordo com a quantidade de focos de zica que visita, utilizando o algoritmo *Get_Places_Amount*.

Algorithm 2: Combinação de Soluções

```

1 Function Combine_Solution( $F, i, G1, G2$ ) :
2    $new \leftarrow \{\}$ 
3    $result \leftarrow G1 \cup G2$ 
4   if  $result.length \neq (i + 1)$  then
5     return
6   else if  $result.length = G1.length$  or  $result.length = G2.length$  then
7      $new \leftarrow result$ 
8   else
9     foreach  $v1 \in G1$  do
10       foreach  $v2 \in G2$  do
11         if  $v1 \in v2.adjacents$  then
12            $new \leftarrow result$ 
13           goto jump
14   jump :
15   if  $new = \{\}$  then
16     return
17    $amount \leftarrow Get\_Places\_Amount(new)$ 
18   if  $new \notin F[i][amount - 1]$  then
19      $F[i][amount - 1].add(new)$ 

```

O algoritmo *Execute* realiza a função principal do problema, no qual preenche cada posição da matriz de soluções $F(v, p)$ combinando valores da iteração anterior.

Para calcular os valores iniciais da matriz de programação dinâmica, o laço de repetição na linha 3 inicializa todos as posições de $F(0, p)$, que são todos os subgrafos conexos $\mathbb{V}' \subset \mathbb{V}$ de no máximo um vértice de tamanho que pode visitar $p + 1$ focos de zica, onde $p \in [0, r - 1]$.

Dentro deste mesmo laço de repetição, na linha 5 temos a verificação de melhor caso, testando todos os vértices do grafo se cada um deles já é uma solução ótima para o problema ou não.

Após o processo inicial, o laço da linha 8 representa a busca por uma solução de tamanho $i + 1$, sendo interrompido ao encontrar uma solução que visite todos os focos de zica. O laço de repetição da linha 9 verifica se $F(i - 1, j)$ possui alguma solução para calcular as combinações de $F(i, j)$, para $j \in [i - 1, r - 1]$.

Em caso positivo, é feita uma combinação de cada subgrafo em presente em $F(i - 1, j)$ para todos os subgrafos presentes em $F(i - 1, k)$, $k \in [i - 1, j]$. Este processo de combinação é feito nos laços de repetição alinhados da linha 15, utilizando o algoritmo *Combine_Solution*.

Nos laços de iteração alinhados da linha 11 fazem a combinação de todos os grafos presentes em $F(i - 1, j)$ com eles mesmos, porém sem calcular combinações repetidas.

No final da iteração iniciada na linha 8, é verificado se há alguma solução em $F(i, j)$, agora que $j = r - 1$. Se houver soluções, elas já são consideradas como ótimas e então o algoritmo *Get_Smaller_Sum* retorna o subgrafo que contém a menor soma de identificadores, resolvendo o problema do ZicaZeroZ.

Algorithm 3: Algoritmo de resolução do ZicaZeroZ

```

1 Function Execute( $\mathbb{V}, \mathbb{R}$ ) :
2    $F \leftarrow \{\{\}\} \times \mathbb{R}.length\} \times \mathbb{V}.length\}$ 
3   foreach  $v1 \in \mathbb{V}$  do
4      $pos \leftarrow Get\_Places\_Amount(\{p\})$ 
5     if  $pos = \mathbb{R}.length$  then
6       return  $\{p\}$ 
7      $F[0][pos - 1].add(\{p\})$ 
8   for  $i \leftarrow 1$  to  $\mathbb{V}.length - 1$  do
9     for  $j \leftarrow (i - 1)$  to  $\mathbb{R}.length - 1$  do
10    if  $F[i - 1][j].length > 0$  then
11      for  $k \leftarrow 0$  to  $F[i - 1][j].length - 2$  do
12        for  $l \leftarrow (k + 1)$  to  $F[i - 1][j].length - 1$  do
13           $Combine\_Solution(F, i, F[i - 1][j][k], F[i - 1][j][l])$ 
14        for  $y \leftarrow (i - 1)$  to  $j - 1$  do
15          for  $k \leftarrow 0$  to  $F[i - 1][j].length - 1$  do
16            for  $l \leftarrow 0$  to  $F[i - 1][y].length - 1$  do
17               $Combine\_Solution(F, i, F[i - 1][j][k], F[i - 1][j][l])$ 
18    if  $F[i][j].length > 0$  then
19      return  $Get\_Smaller\_Sum(F[i][j])$ 
20   return  $\{\}$ 

```

3. Análise da Complexidade

As verificações que serão feitas nesta seção trata-se de uma execução do pior cenário possível para este algoritmo, tendo v vértices, v^2 arestas, v focos de zica, sendo que cada vértice visita apenas um foco.

3.1. Análise do algoritmo Get_Smaller_Sum

O algoritmo *Get_Smaller_Sum* recebe por parâmetro uma lista de subgrafos L , onde $\{\forall x \in L, x \subset \mathbb{V}\}$ e retorna o subgrafo que contém a menor soma dos identificadores de seus vértices.

É necessário acessar todos os vértices de todos subgrafos da lista para fazer a soma acumulada depois e realizar uma busca linear pelo menor número. Porém, no pior caso, esta função executará apenas v verificações, pois o único subgrafo possível de solução é o próprio grafo. Logo:

$$O(\mathbb{V}) \quad (1)$$

Para a complexidade de espaço, este algoritmo utiliza um vetor de números inteiros para contabilizar os identificadores. No pior caso este vetor tem tamanho máximo de \mathbb{V} .

$$O(\mathbb{V}) \quad (2)$$

Tabela 1. Complexidade do algoritmo Get_Smaller_Sum

Complexidade de Tempo	Complexidade de Espaço
$O(\mathbb{V})$	$O(\mathbb{V})$

3.2. Análise do algoritmo Get_Places_Amount

O algoritmo faz um percurso linear entre os vértices do subgrafo que for recebido por parâmetro. No pior caso, o subgrafo pode ser o próprio grafo do problema. Portanto a complexidade de tempo será:

$$O(\mathbb{V}) \quad (3)$$

Para a complexidade de espaço, neste algoritmo é criada uma lista para armazenar os focos de zica visitados pelos vértices para que possa ser calculado seu tamanho posteriormente. No pior caso, essa lista terá o tamanho de r , que é a quantidade de focos de zica do problema.

$$O(\mathbb{R}) \quad (4)$$

Tabela 2. Complexidade do algoritmo Get_Places_Amount

Complexidade de Tempo	Complexidade de Espaço
$O(V)$	$O(R)$

3.3. Análise do algoritmo Combine_Solution

O algoritmo faz a combinação de dois subgrafos conexos de V em apenas um subgrafo, porém garantindo que seja conexo.

Para fazer essa verificação, é necessário navegar para cada vértice de um subgrafo, em busca de uma aresta para o outro subgrafo. Apesar de muito improvável, o pior caso desta verificação é:

$$O(V^2E) \quad (5)$$

Para a complexidade de espaço, o subgrafo resultante da combinação no pior caso pode ser o próprio grafo do problema, portanto a complexidade de espaço do algoritmo será:

$$O(V) \quad (6)$$

Tabela 3. Complexidade do algoritmo Combine_Solution

Complexidade de Tempo	Complexidade de Espaço
$O(V^2E)$	$O(V)$

3.4. Análise do algoritmo Execute

O algoritmo começa realizando uma iteração na lista de vértices do grafo para criar as soluções de $F(0, p)$, tendo um custo de apenas $O(1)$ do algoritmo *Get_Places_Amount*, pois o subgrafo passado por parâmetro possui apenas um vértice.

Apesar de todos os subgrafos criados serem colocados na posição $F(0, 0)$ da matriz, este laço ainda tem um custo de V .

Após isso, temos o laço que irá caminhar entre as posições da matriz de soluções F . Porém, no pior caso proposto não terá soluções válidas para as posições $F(i, j)$ onde $i \neq j$, pois no pior caso sempre v vértices visitam v focos de zica. Portanto, o laço exterior junto ao laço interior irão executar VR vezes, porém apenas as posições $i = j$ terão algum processamento.

Cada combinação feita terá um custo de $w(w - 1)/2$ chamadas de *Combine_Solution*, onde w é o número de subgrafos na posição $F(i - 1, j - 1)$. No total, irão ser gerados 2^v subgrafos, ou seja, todos os subgrafos serão gerados caso o problema seja o pior caso.

$$O(2^V) \quad (7)$$

A complexidade de espaço é grande quando observamos que em cada posição $i = j$ da matriz de soluções F temos uma lista de subgrafos de \mathbb{G} . Novamente, o pior caso é improvável, porém cada posição da matriz $\mathbb{V} \times \mathbb{R}$ poderia armazenar até \mathbb{V} .

Tabela 4. Complexidade do algoritmo Execute

Complexidade de Tempo	Complexidade de Espaço
$O(2^{\mathbb{V}})$	$O(2^{\mathbb{V}})$

4. Características da Implementação

A implementação foi feita utilizando a linguagem *Python*. Os vértices foram modelados na classe chamada *Person*, e o problema pode ser executado várias vezes, pois também está modelado como uma classe chamada *Problem*.

O arquivo *main.py* faz a leitura do arquivo de entrada, criando uma lista com as instâncias de *Person* e criando as relações entre elas usando uma lista de adjacência em cada objeto.

O arquivo *helper.py* contém a implementação das classes *Person* e *Problem*. O método *Problem.execute()* inicia a execução do algoritmo, utilizando os dados atuais de instâncias *Person*.

Mais detalhes técnicos foram inseridos nos comentários do código-fonte.

5. Execução de Testes

Os testes foram focados utilizando instâncias de pior caso com grafos densos e depois com grafos esparsos.

Foram criadas sete instâncias que representam o pior caso, para cada uma delas sendo v vértices, onde cada vértice visita apenas um foco de zica, v^2 arestas e tamanho da solução v . Foram criadas instâncias de tamanho [5, 11] pois neste intervalo apresentam uma mudança significativa de comportamento. Na Figura 1 é apresentado o gráfico de tempo de execução destas instâncias.

No segundo teste, foram criadas sete instâncias de pior caso novamente, porém a o tamanho da solução será de $\frac{v}{2}$ vértices. Na Figura 2 mostra que o comportamento foi exatamente o mesmo, só o que o número de instâncias agora foram no intervalo de [6, 12], que é um vértice à mais do teste anterior.

Isso mostra que o tempo de pior caso do algoritmo relaciona o número de vértices com o tamanho da solução para gerar seu custo.

O último teste também se caracteriza como pior caso, porém tem um custo mínimo de arestas, para cada instância do problema com v vértices, visitando cada um apenas um foco de zica, temos $v - 1$ arestas, entretanto o tamanho da solução do problema ainda é v .

No teste da Figura 3 mostra que o custo segue um padrão polinomial, muito menor do que o padrão apresentado por grafos muito densos.

6. Comparação entre a Análise e os Testes

A análise teórica mostrou que o algoritmo de resolução baseado em programação dinâmica tem um custo pseudo-polynomial no caso médio, apesar de ser um polinomial

Figura 1. Gráfico de execuções com grafos densos com $S = \mathbb{V}$

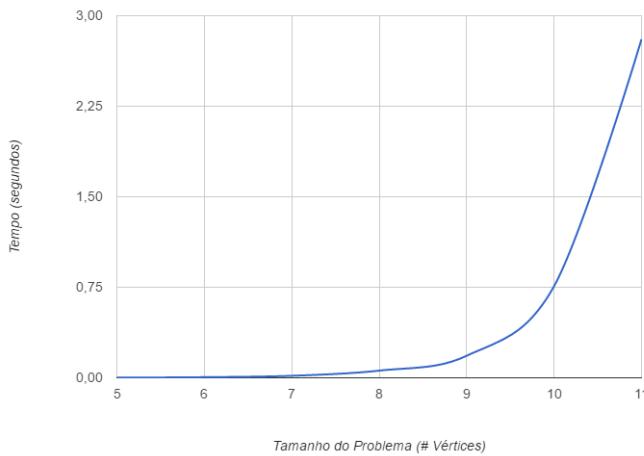


Figura 2. Gráfico de execuções com grafos densos com $S = \mathbb{V}/2$

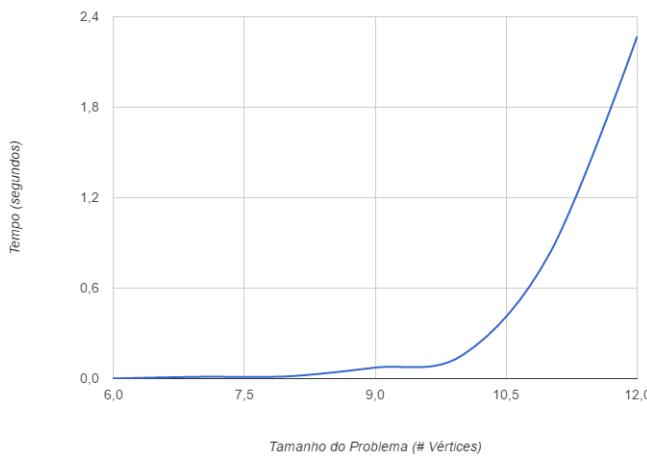
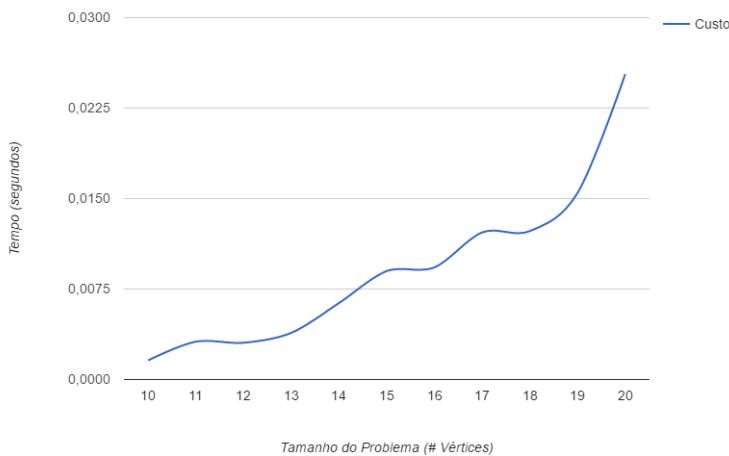


Figura 3. Gráfico de execuções com grafos esparsos com $S = \mathbb{V}$



alto, porém é exponencial no pior caso. A quantidade de arestas está diretamente relacionada ao custo geral do algoritmo, pois seu tamanho varia muito de instância para instância.

Relacionando o primeiro e segundo teste realizados com a complexidade analisada na Seção 3 temos que os resultados são bastante semelhantes.

O gráfico gerado pela função pura 2^V mostra um comportamento muito parecido com os obtidos nos testes de pior caso para grafos densos.

Para a verificação do teste de pior caso para grafos esparsos, porém tendo como tamanho da solução $S = \mathbb{V}$, os resultados foram muito melhores, se aproximando ao custo polinomial.

Referências

[Cormen, Thomas H. et al. 2009] Cormen, Thomas H. (2009). *Introduction to Algorithms, Third Edition.*

Trabalho Prático: ZikaZeroZ

Felipe Gomes de Oliveira

¹Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

{felipe.inad}@gmail.com

1. Exercício 1

Um grafo **G** é constituído de um conjunto **V** não-vazio de elementos, denominados vértices, e um conjunto **A** de pares não ordenados de elementos de **V**, denominados arestas. A denotação de grafo é dada por **G(V, A)** ou simplesmente **G**. Para o problema abordado neste trabalho, é considerado um grafo **G**, onde **V** representa o conjunto de voluntários para procurar e eliminar os focos de zika. **A** corresponde ao conjunto de relacionamentos (Amizade) entre os voluntários.

Para atender o contexto do trabalho foi definido o conjunto de focos de reprodução do mosquito zika, sendo este conjunto representado por **F**. Para a realização da tarefa de busca e eliminação dos focos foi definida a relação, **R(v): V → F**, entre os voluntários que conseguem verificar os focos de reprodução. Esta relação entre voluntários e focos é representada no grafo por meio da apresentação do conjunto de focos de zika que podem ser encontrados por cada voluntário.

Para este trabalho foram feitas as seguintes suposições: *a)* o grafo é não-dirigido; *b)* o grafo é não ponderado; e *c)* o grafo é conexo. Também é apresentada a restrição de que somente será admitido como solução, o conjunto de vértices (voluntários) que formarem um grafo conexo.

2. Exercício 2

Para resolver o problema proposto neste trabalho o grafo modelado na questão anterior foi implementado em uma matriz de adjacências, contendo os vértices e as arestas. Outra matriz foi criada para armazenar a relação entre os voluntários e os focos que podem ser alcançados por cada voluntário. Em seguida, é executada a operação que irá verificar todas as possibilidades de voluntários que podem alcançar todos os focos de reprodução.

A operação de verificação, inicialmente, cria uma sequência com todas as possibilidades de voluntários, em ordem crescente. Em seguida cada conjunto de voluntários candidato é confrontado com a matriz de focos, no intuito de verificar se estes voluntários conseguem cobrir todos os focos de reprodução zika. Posteriormente, é verificado se o conjunto de voluntários alcança todos os focos e em seguida é analisado se os vértices (voluntários) estão conectados. Se um conjunto candidato de voluntários, que satisfaça todas as suposições e restrições for encontrado a busca para imediatamente, encerrando a execução do problema.

Para verificar se o grafo é conexo, foi implementado um algoritmo baseado na busca em largura. Para isso, a busca é executada para cada vértice, sendo atribuído um rótulo (cor) que não permitirá que um determinado vértice seja acessado mais de uma vez. Dessa forma, caso o grafo não seja conexo será incrementado um contador que computa a quantidade de componentes desconexos do grafo.

3. Exercício 3

A complexidade temporal do algoritmo proposto pode ser inferida por meio da análise de cada segmento da implementação. Para a inicialização das matrizes obtem-se a complexidade: $O(N^2)$, já que o mesmo deve percorrer toda a matriz para inicializá-la. Em seguida, para a criação do conjunto de possibilidades de voluntários, foi proposta uma estratégia de alta complexidade computacional, sendo obtida a complexidade N^n . Sendo que, mesmo que a ordem das possibilidades seja disposta em ordem crescente e que a busca pare ao encontrar o menor valor, ainda assim o problema de alta complexidade se mantém.

Para verificar se um determinado conjunto consegue cobrir todos os focos de reprodução, são percorridos os N^n conjuntos e os vértices dos conjuntos são confrontados com os focos da Matriz de Focos, obtendo complexidade de: $O(N^n + (N+F))$. Para verificar se os grafos candidatos são conexos foram aplicadas N buscas em largura por conjunto, alcançando complexidade de $O(N^n + N(N+M))$. Após a análise, é possível constatar que a complexidade temporal do algoritmo é dominada pelo processamento das N^n combinações, resultando em um complexidade de $O(N^n)$.

Considerando a complexidade espacial do algoritmo proposto, devem ser consideradas as matrizes de adjacências e de focos, onde a complexidade espacial da matriz de adjacências é $O(N^2)$. Enquanto que a complexidade espacial da matriz de focos é definida por $O(N.F)$.

4. Exercício 4

O algoritmo proposto como solução do problema apresentado neste trabalho, foi implementado na linguagem C. O diretório ZikaZeroZ, contém os arquivos: compilar.sh, executar.sh, TrabalhoGrafos.c, os arquivos de entrada fornecidos pelo monitor da disciplina (in0 até in7) e os arquivos de entrada de 5 exemplos solicitados no Exercício 5.

Para a compilação e execução do programa, como definido na especificação do trabalho, deve ser executado o arquivo compilar.sh, da seguinte forma: ./compilar.sh. Em seguida deve ser executado o arquivo executar.sh da seguinte forma: ./executar.sh in0 out0. Onde será realizada a leitura do arquivo de entrada in0 e será produzido um arquivo de saída, com a solução do problema para aquela configuração.

5. Exercício 5

Para experimentar a abordagem apresentada, foram propostos alguns exemplos. O primeiro exemplo é composto por 8 vértices, 8 arestas e 10 focos, como mostrado na Figura 1 abaixo. Para este exemplo foi apresentado o conjunto solução: 2, 5, 6, 7 e 8.

Para o segundo exemplo, é apresentado um grafo com 8 vértices, 7 arestas e 9 focos, como pode ser visto na Figura 2. Para este exemplo é apresentado o conjunto solução: 1, 2, 3, 6, 7 e 8.

No terceiro exemplo, é apresentado um grafo com 6 vértices, 7 arestas e 6 focos, como pode ser visualizado na Figura 3. Para este exemplo é apresentado o conjunto solução: 2 e 3.

Para avaliar o desempenho temporal da abordagem proposta foram realizados experimentos variando o número de vértices, partindo de 3 até 14 e fixando a quantidade de arestas em função da quantidade de vértices (N), ou seja, $(N^2-N)/4$ e a quantidade de focos em $N^2/2$. Para cada variação de vértices, foram realizadas 10 execuções, sendo então

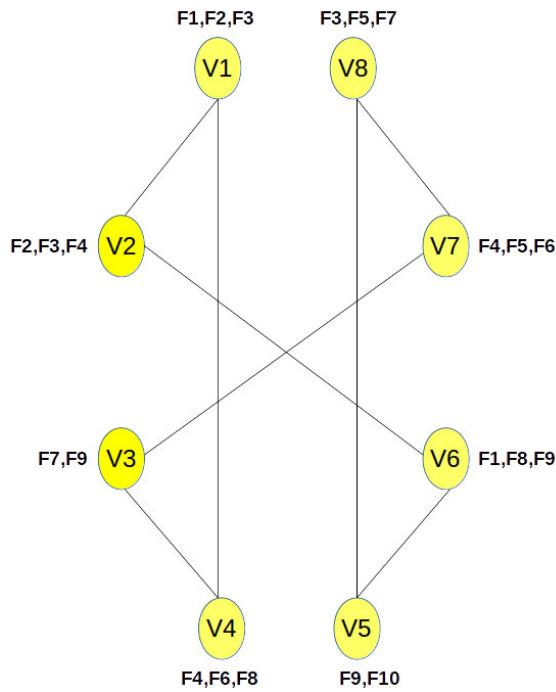


Figura 1. Exemplo 1

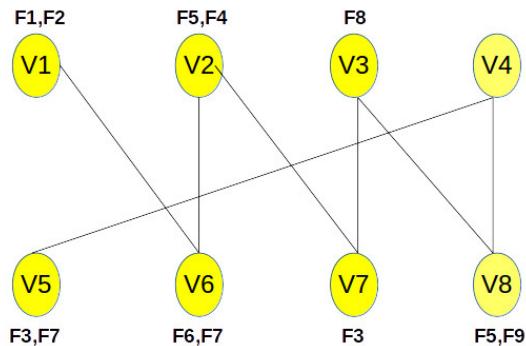


Figura 2. Exemplo 2

tirada a média das 10 para representar o tempo de execução para um número específico de vértices. No gráfico abaixo é possível observar o rápido e acentuado crescimento do tempo de execução para um número maior de vértices, já que o cálculo das combinações é baseado no número de vértices. A partir de 15 vértices o tempo de execução aumenta substancialmente, até inviabilizar a execução.

6. Exercício 6

O tempo de execução teórico previsto para a execução do algoritmo proposto apresenta complexidade de $O(N^n)$, que é obtida por meio do processo de criação das possibilidades de voluntários para cobrir todos os focos de reprodução do mosquito zika. Após a realização dos experimentos e analisando a curva de crescimento temporal, é possível constatar a conformidade entre a análise teórica e os resultados alcançados durante o processo experimental. Já que ambas as análises convergem para o mesmo comportamento durante a execução dos experimentos.

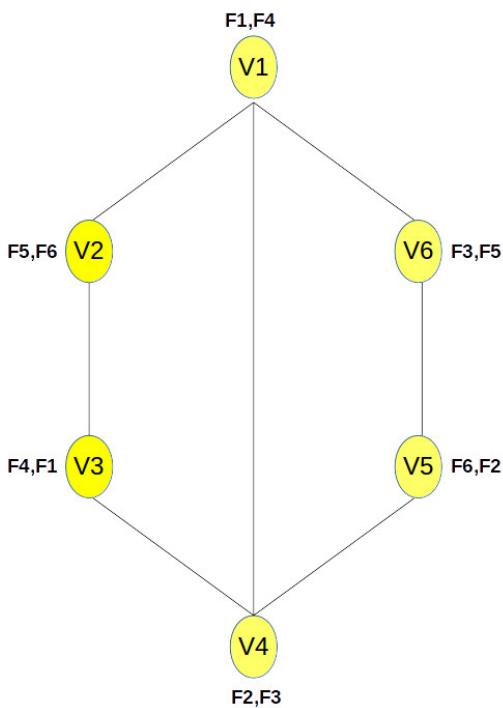


Figura 3. Exemplo 3

No entanto, embora o desempenho temporal seja insatisfatório, os resultados obtidos pelo algoritmo são ótimos. Dessa forma, foi alcançado o objetivo do trabalho, que previu a implementação de algoritmos não-ótimos, porém devem apresentar soluções ótimas.

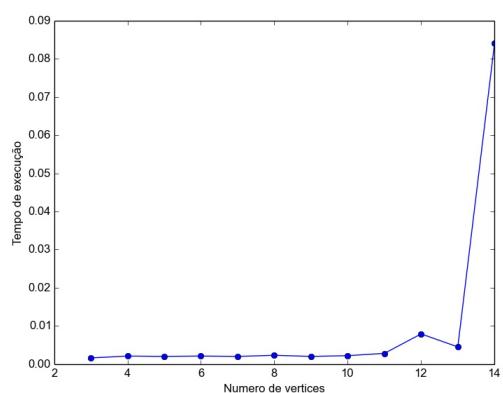


Figura 4. Gráfico 1

Relatório do trabalho prático 1 - Grafos

Rodrigo Lemos Cardoso

¹Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

rodrigolc@dcc.ufmg.br

1. Modelagem

Para este trabalho escolheu-se uma modelagem de grafo não dirigido que classifica os vértices em dois tipos:

- Voluntários - São vértices que conectam-se entre si formando a rede de pessoas que se conhecem cujo relacionamento é dado pela aresta.
- Focos do Mosquito - São vértices que se conectam apenas a vértices do tipo Voluntário, pois a aresta entre os dois representa o fato daquele voluntário ser capaz de alcançar diretamente aquele foco do mosquito.

Desse modo a figura 1 dá um exemplo da representação em grafos para este problema. Os vértices a esquerda representam os voluntários e podem estar interconectados. Os vértices a direita são os focos de mosquito e são alcançáveis diretamente apenas por voluntários.

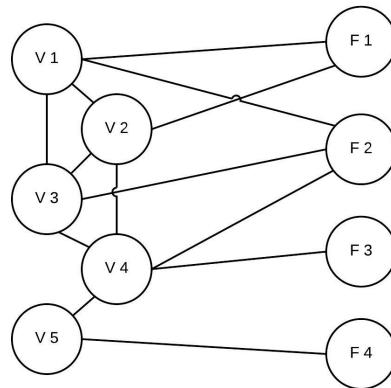


Figura 1. Representação em grafo do problema

Algumas restrições devem ser tomadas:

- O grafo inicial precisa ser conexo;
- Precisa-se de pelo menos um usuário e um foco;
- O problema deve ser solucionável, isto é, todos os focos devem ser alcançáveis por algum subconjunto de usuários no grafo.

2. Algoritmo

Para resolver este problema adotou-se uma abordagem força bruta para buscar todas as soluções possíveis e então retornar a solução de menor tamanho.

Primeiramente é gerado o grafo de relações entre voluntários e focos em um grafo não dirigido. Depois um laço itera uma variável k de 1 até o número de voluntários gerando em cada iteração todas as soluções possíveis de tamanho k .

Cada solução gerada será um conjunto de vértices de voluntários candidato a solução e para ser uma solução deverá atender a dois requisitos:

- Formar um conjunto conexo de voluntários;
- Cobrir todos os focos

O procedimento de geração de soluções de tamanho k será dado da seguinte maneira: Duas chamadas recursivas serão feitas a princípio, uma colocando o primeiro voluntário na solução e outra não o colocando. Cada uma das chamadas fará a mesma coisa para o segundo usuário, fazendo novamente uma chamada o incluindo e outra não o incluindo. Segue-se sucessivamente este procedimento até que k usuários sejam colocados na solução (Um pequeno tratamento é feito para que 1 usuários sejam sucessivamente escolhidos caso cheguemos em um ponto onde $k-1$ foram escolhidos e restam 1 usuários para se analisar).

O seguinte algoritmo resolve o problema de cobrir todos os focos de mosquito com o menor número de voluntários possível em uma rede de contatos conexa:

Algorithm 1 Algoritmo principal

```
1: procedure ZIKAZEROZ( $G, V, F$ )       $\triangleright$  menor subconjunto conexo de  $V$  que cobre  $F$ 
2:    $k \leftarrow 0$ 
3:   while  $k \neq |V|$  do                       $\triangleright$  Itera até o número de voluntários
4:      $S \leftarrow \text{GenerateSolutions}(G, V, k)$      $\triangleright S$  recebe uma solução de tamanho  $k$ 
5:     if  $S$  is not empty then                   $\triangleright$  Se foi encontrado um subconjunto
6:       break
7:     end if
8:      $k+ = 1$ 
9:   end while
10:  return  $S$                                  $\triangleright$  Menor subconjunto
11: end procedure
```

Algorithm 2 Gerador de soluções

```
1: procedure GENERATESOLUTIONS( $G, V, k$ )  $\triangleright$  Retorna uma solução de tamanho  $k$ 
2:    $vec \leftarrow \{1..|V|\}$ 
3:   for  $s \in$  permutations of  $vec$  of size  $k$  do  $\triangleright$  Escolhemos  $k$  elementos de  $vec$ 
4:     if  $V[s]$  is connected AND  $V[s]$  covers  $F$  then
5:       break
6:     end if
7:   end for
8:   return  $s$   $\triangleright$  Subconjunto de tamanho  $k$ 
9: end procedure
```

Observe que na linha 3 do Algoritmo 2 são geradas soluções de tamanho k para um vetor de voluntários de tamanho $|V|$. Estes valores são gerados de modo ordenado desde o menor índice para o maior índice. Por exemplo, em 5 voluntários deseja-se tomar 3 voluntários destes. Serão gerados respectivamente:

$$\{1,2,3\}; \{1,2,4\}; \{1,2,5\}; \{1,3,4\}; \{1,3,5\}; \{1,4,5\}; \{2,3,4\}; \{2,3,5\}; \{2,4,5\}; \{3,4,5\}$$

Estes valores são gerados de forma recursiva e armazenados em uma pilha. O último elemento do vetor é o topo da pilha, quando o rejeitamos para gerar outra solução nós o removemos e inserimos o valor seguinte. Funciona como uma árvore binária balanceada de tamanho $|V|$ e $2^{|V|} - 1$ vértices. A aresta da esquerda de cada vértice significa que ele foi inserido na pilha e a aresta a direita significa que ele não foi inserido e, portanto, desempilhado. A árvore é então percorrida começando pela esquerda e indo à direita.

A linha 4 verifica se a solução candidata resolve o problema. É feita uma busca em largura em um dos vértices da solução candidata no subgrafo que esta representa. Se todos os voluntários da solução forem alcançáveis e todos os focos forem cobertos então esta é uma solução para o problema.

A solução ótima ordenada será sempre a primeira a ser encontrada, pois os subconjuntos são checados de maneira ordenada e sempre do menor subconjunto ao maior subconjunto.

Observação: Não é do escopo deste trabalho mas no código foi implementada uma heurística que calculava um limite superior e um limite inferior para os valores de k . Para os testes não foi usada a heurística.

3. Complexidade

A complexidade será dada pelo número de chamadas ao algoritmo de verificação multiplicada pela complexidade dessa verificação.

A verificação é feita em tempo $k + p + |F|$ onde k é o tamanho da solução verificada (tempo para percorrer a lista de adjacência de k voluntários).

O valor p é dado pelo número de arestas m do grafo original multiplicado pela fração $\frac{k}{n}$

de voluntários do total.

No algoritmo principal é sabido que são feitas $|V|$ chamadas ao gerador de solução com valores k variando de 1 até $|V|$. Portanto, a função de complexidade é dada por:

$$\sum_{k=1}^{|V|} \binom{|V|}{k} (k + p + |F|) = O(2^{|V|}(|V| + |F|)) \text{ pois:}$$

$$\sum_{k=1}^{|V|} \binom{|V|}{k} (k) = 2^{|V|-1}(|V|)$$

$$\sum_{k=1}^{|V|} \binom{|V|}{k} (p) = O(2^{|V|}(|V|)) \text{ pois } p \text{ será fixo em } 4|V|$$

$$\sum_{k=1}^{|V|} \binom{|V|}{k} (|F|) = 2^{|V|}(|F|)$$

A complexidade de espaço é linear com o número de vértices e arestas do grafo. Nenhuma estrutura de tamanho exponencial é armazenada.

4. Implementação

Os algoritmos foram implementados na linguagem de programação C++ (11) e estão anexados juntos a este documento.

5. Testes

Teste foram feitos variando-se o tamanho do problema. Como se trata de um problema de ordem de complexidade assintótica exponencial, optou-se por gerar testes com valores pequenos. Variou-se o número de voluntários de 20 a 140 aumentando os valores de 20 em 20. Variou-se a quantidade de focos em: um oitavo, um quarto e metade do número de voluntários.

Para cada instância foi gerada uma “malha” retangular de voluntários onde os voluntários das quinas se conectavam apenas a 2 voluntários. Os das bordas não-quinas se conectavam a 3 e os demais se conectavam aos 4 voluntários em torno desses.

Os focos eram distribuídos aleatoriamente de modo que cada voluntário possuísse uma quantidade aleatória de focos (limitada pela metade do número total de focos).

Os testes foram feitos em um intel core i7 3.6 GHz, 4GB de RAM em um Ubuntu. Para validade estatística repetiu-se cada medida 30 vezes e tomou-se a média de execução e o desvio padrão. (Alguns resultados que estão omissos nos gráficos mostraram-se inviáveis de se computar)

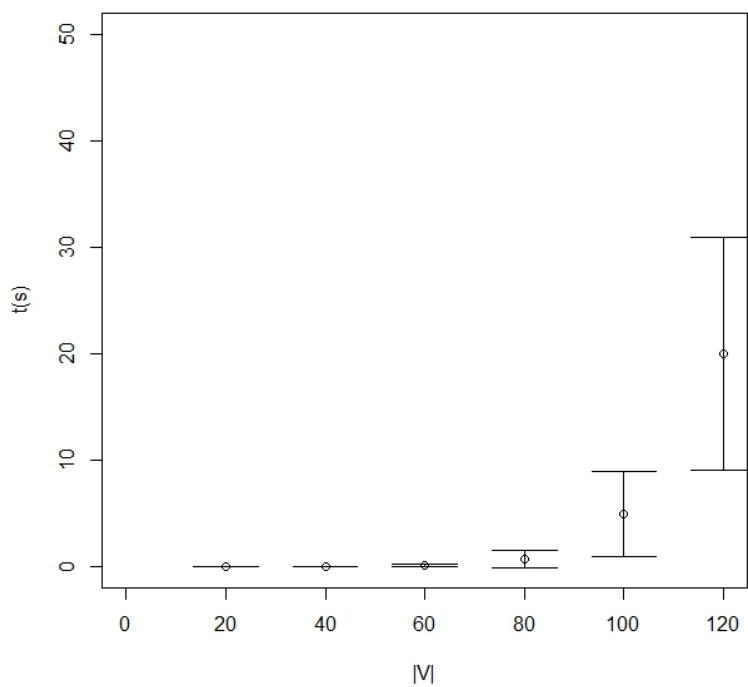


Figura 2. Número de focos igual a um oitavo do número de voluntários

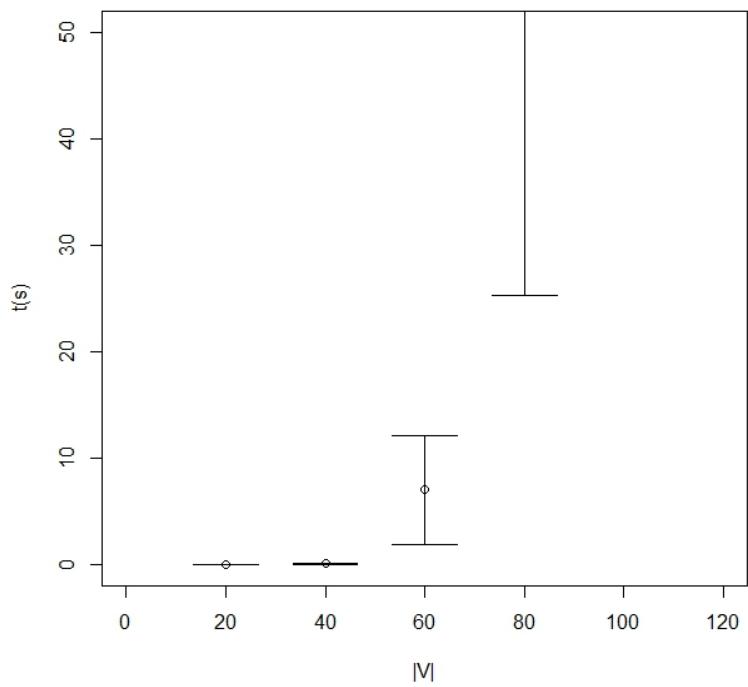


Figura 3. Número de focos igual a um quarto do número de voluntários

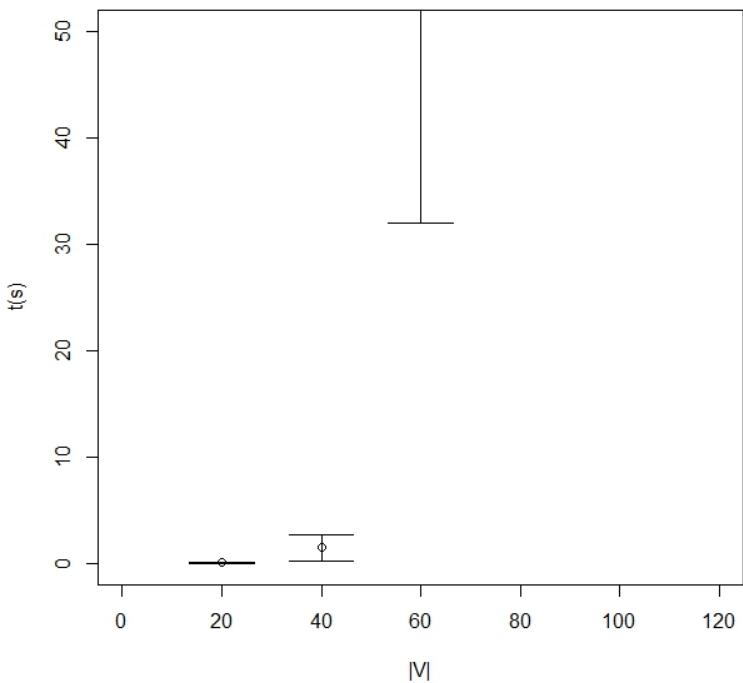


Figura 4. Número de focos igual a metade do número de voluntários

6. Comparação

Como foi enunciado teoricamente, é um problema que leva tempo exponencial para ser resolvido. Mais ainda, multiplicado pelo número de vértices no grafo. Na prática foi constatado que para certos tamanhos de entrada é inviável utilizar um algoritmo força bruta. Um caso bem evidente disso é o resultado mostrado na figura 3, onde um aumento na entrada cresceu muito o tempo de execução (o resultado para o 80 tomava mais de 1 dia).

Observa-se o quanto o número de focos influencia no tempo de execução pois este fará a checagem da solução ser cada vez mais demorada. Como a verificação é feita apenas por uma busca em largura que colore os vértices de foco na medida em que os voluntários candidatos são descobertos assume-se então que a influência desses valores é linear no tempo de verificação.

Nota-se um desvio padrão muito grande em cada resultado. Isto se deve ao fato de que instâncias mesmo grandes podem ser muito propícias a ter o ótimo calculado de maneira eficiente ou ineficiente. Por exemplo, a instância pode ter tamanho 1.000.000.000 e 1.000.000 de focos. Se os voluntários 1,2,3,...,1.000.000 já forem capazes de cobrir todos os focos, a solução ótima é calculada instantaneamente. No pior caso para isso não haverá tempo nem máquinas o suficiente para se computar o resultado.

Trabalho Prático de Grafos - PAA

Leandro T. C. Melo

¹DCC - UFMG

ltcmelo@dcc.ufmg.br

Resumo. Este é o relatório do Trabalho Prático de Grafos, da disciplina PAA. Ele consiste da resolução do Problema ZikaZeroZ e exercícios relacionados.

1. Exercício 1

Uma modelagem para o problema é a seguinte. A partir das informações de voluntários e os focos aos quais eles tem acesso, construímos um grafo H . Esse é um grafo bipartido, onde os voluntários representam um conjunto, X , e os focos representam o outro conjunto, Y . Toda aresta em H conecta um vértice de X a um vértice de Y , correspondendo a noção de *acesso* ao foco.

Nosso objetivo é encontrar o menor número de vértices de X que *cobre* todos os vértices de Y . Esse é um problema clássico de grados conhecido como *set cover*. No caso, os focos, correspondentes ao conjunto Y do grafo bipartido, representam o conjunto universo, U . Os voluntários, correspondentes ao conjunto X do grafo bipartido, representam uma *família* de subconjuntos, S , do conjunto U . A união de todos os subconjuntos de S é igual a U . Portanto, para encontrarmos o menor número de voluntários com acesso a todos os focos podemos calcular o set cover do grafo bipartido. Garantimos que todo foco é acessado por pelo menos um voluntário.

No entanto, há ainda uma exigência no problema que esse set cover, o conjunto de voluntários selecionada, forme um grafo conexo quando induzido no grafo G , que é o grafo "primário" do problema que representa os voluntários como vértices e suas relações de amizade como arestas. Dessa forma, é possível que o set cover encontrado não atenda a esse requisito. Então, a alternativa é calcular não apenas o menor set cover, mas todos os possíveis set cover e ordená-los em função daqueles com o menor número de vértices (conforme exigido pelo problema, se dois ou mais set cover contiverem o mesmo número de vértices, então eles devem ser ordenados pelos índices dos voluntários).

Tendo uma sequência ordenada em função dos menores set cover para os maiores set cover, podemos testá-la iterativamente até que encontremos um set cover tal que seus vértices formem um grafo induzido conexo em G . O set cover encontrado é a solução para o problema. Se chegarmos até o final dessa sequência sem que algum set cover válido, perante o critério de conectividade, seja encontrado, então não haverá solução para o problema.

2. Exercício 2

Encontrar um set cover é um problema difícil e os algoritmos eficientes conhecidos são apenas aproximações. Mas neste TP é exigida a solução ótima. Logo, a estratégia utilizada é a de força-bruta. O algoritmo para resolução do problema, `FindZikaZeroSolution` é descrito abaixo em pseudo-código. Ele utiliza dois outros algoritmos, `ComputeSetCover` e `CheckConnectivity`, os quais

também são descritos abaixo. A função `genPowerSet`, chamada pelo algoritmo `ComputeSetCover` não é mostrada, pois se trata de uma função genérica para geração de um *power set* de um conjunto de números inteiros que pode ser encontrada em inúmeras bibliotecas de software - nela, não há nada específico a grafos. O pseudo-código dos algoritmos é um mapeamento fidedigno da implementação realizada em C++ e pode ser verificado a partir do código-fonte submetido.

Algorithm 1 FindZikaZeroSolution

Require: Grafo G, de V voluntários e A arestas de amizade.

Require: Conjunto F de r focos e uma relação de $V \rightarrow F$.

Ensure: Menor conjunto V' de voluntários, tal que todo r é acessado e o grafo induzido por V' em G é conexo.

```

1: Crie um grafo H, de r vértices (os focos).
2: Adicione V vértices em H, mantendo uma mapa, M, de correspondência entre esses
   vértices e os vértices de voluntários em G.
3: SetCoverList  $\leftarrow ComputeSetCover(H, M)$ 
4: for SetCover  $\in$  SetCoverList do
5:   IsConnected  $\leftarrow CheckConnectivity(G, M, SetCover)$ .
6:   if not IsConnected then
7:     Descarte o SetCover, pois não leva a grafo induzido conexo.
8:   else
9:     Winner  $\leftarrow$  SetCover // Vértices de H são substituídos pelos de G através do
   mapa M.
10:    return Winner.
11:   end if
12: end for
```

3. Exercício 3

Será feita uma *análise agregada*, assim como nos capítulos de grafos do livro do Cormen. O algoritmo `ComputeSetCover` introduz um fator de 2^N onde $N = |V|$ (devido à geração do próprio power set). Este mesmo algoritmo percorre as adjacências de todos os subconjuntos do power set calculados com o intuito de verificar o acesso aos focos no grafo bipartido. Isso introduz um fator V'' que corresponde às adjacências desses subconjuntos, as quais são variáveis em função das arestas originais do grafo G. Portanto, uma análise precisa para esse fator é difícil. Mas ainda assim, há um ordenação ao final ainda do algoritmo `ComputeSetCover` para ordenar os conjuntos de acordo com o critério exigido pelo enunciado. Tal ordenação suprirá, de maneira ou de outra, o fator variável mencionado acima. Dessa forma, a complexidade de `ComputeSetCover` é dada por $O(2^V \cdot \lg 2^V) = O(2^V \cdot V \lg 2) = O(V \cdot 2^V)$ - o algoritmo de ordenação é o quicksort. A partir do cálculo dos vértices candidatos (implicados pelos subconjuntos do power set), o algoritmo principal, `FindZikaZeroSolution`, realiza um teste de conectividade para cada um deles. Esse teste é realizado através do algoritmo `CheckConnectivity` o qual é basicamente uma DFS. Em princípio todos os subconjuntos do power set podem ser candidatos e serão testados (até o último deles) por conectividade. Sendo assim, a complexidade do algoritmo principal incorpora o fator de $O(V + E)$ (o grafo é representado por uma lista de adjacência) e tem a complexidade

Algorithm 2 ComputeSetCover

Require: Grafo bipartido H, onde o conjunto universo são os focos e os subconjuntos de famílias são os voluntários com acesso a determinados focos.

Require: Mapa de correspondência M de vértices entre o grafo H e do grafo G.

Ensure: Uma lista de set covers ordenadas dos menores aos maiores set covers, sendo que set covers do mesmo tamanho são ordenados pelos índices de seus vértices.

```
1: Candidates ←  $\emptyset$ 
2: TotalFocuses ← H.numVertices – M.length
3: PowerSet ← GenPowerSet(M.length)
4: for Subset ∈ PowerSet do
5:   FoundFocuses ←  $\emptyset$ 
6:   for v ∈ Subset do
7:     for u ∈ H.adjVertices(v) do
8:       if u ∉ FoundFocuses then
9:         FoundFocuses ← FoundFocuses  $\cup$  u
10:      end if
11:    end for
12:  end for
13:  if FoundFocuses == TotalFocuses then
14:    Candidates ← Candidates  $\cup$  Subset
15:  end if
16: end for
17: Sort(Candidates) // De acordo com o critério mencionado acima.
18: return Candidates
```

Algorithm 3 CheckConnectivity

Require: Grafo G, de V voluntários e A arestas de amizade.

Require: Mapa de correspondência M de vértices entre o grafo H e do grafo G.

Require: Conjunto de vértices V' de G a serem testados por conectividade induzida.

Ensure: Se o grafo induzido por V' em G é conexo.

```
1: // O Algoritmo é basicamente uma DFS de pilha explícita e que percorre apenas
   aquelas adjacências oriundas do conjunto de vértices V'.
2: v ← V'[0]
3: Stack ← ∅
4: Push(Stack, v)
5: Visited ← ∅
6: Visited ← Visited ∪ v
7: PowerSet ← GenPowerSet(M.length)
8: while not Stack.empty do
9:   u ← Pop()
10:  for w ∈ H.adjVertices(u) do
11:    if Visited[w] == True then
12:      continue
13:    end if
14:    if w ∉ V' then
15:      continue
16:    end if
17:    Visited ← Visited ∪ w
18:    Push(Stack, w)
19:  end for
20: end while
21: if Visited == M.Keys then
22:   return True // Os vértices do subconjunto estão conectados.
23: end if
24: return False
```

final de $O(2^V \cdot (V + E)) = O(V \cdot 2^V + E \cdot 2^V) \approx O(V \cdot 2^V)$, caso estejamos lidando com grafos esparsos, como foi feito nos experimentos deste TP. A complexidade espacial é $O((V + E) \cdot 2^{V+E})$ devido ao armazenamento dos grafos que correspondem aos subgrafos do conjunto power set do grafo original.

4. Exercício 4

A implementação do algoritmo proposto foi realizada em C++ e todo código-fonte segue em anexo. Sua documentação está embutida como comentários no próprio código-fonte. Vale re-lembrar que a implementação não é eficiente. Mesmo sendo baseada em uma estratégia de força-bruta, ela poderia tirar proveito de certas características do problema. Em particular, a implementação sempre gera todo o conjunto power set e sempre percorre todas adjacências do grafo bipartido antes de iniciar os testes de conectividade. Mas ela poderia, em princípio, realizar esse processo iterativamente e, assim que o resultado ótimo fosse encontrado, não haveria necessidade de testar outros subconjuntos do power set (e obviamente percorrer suas adjacências para detectar a presença de todos os focos). O algoritmo apresentado e a implementação podem ser facilmente ajustados, pois o código é modular. No entanto, como a eficiência não está sendo avaliada aqui, tal abordagem foi escolhida para deixar mais clara e conveniente a análise de complexidade. NOTA: No arquivo algo.h há uma flag para habilitar/desabilitar o *trace* da evolução do algoritmo. Caso necessário, ela pode ser habilitada para inspeção mais detalhada de cada um dos passos.

5. Exercício 5

Foram realizados os seguintes testes de execução:

- O número de vértices do grafo de voluntários foi gradativamente aumentado, tendo como ponto de partida o próprio exemplo do enunciado do TP, sendo que a última execução foi em um grafo com 23 vértices de voluntários. O número de arestas também sofreu um aumento proporcional ao aumento do número de vértices, especificamente de forma tal que $|E| \approx 2|V|$. O aumento do número de focos foi feito de maneira a permanecer $|r| \approx |V|/2$. A curva do tempo de execução para esses testes é mostrada na Figura 1.
- O número de vértices e arestas do grafo de voluntários foi aumentado exatamente da mesma maneira que o item anterior. No entanto, o número de focos foi aumentado de maneira tal que o $|r| \approx |V|$. A curva do tempo de execução para esses testes é mostrada na Figura 2.
- A execução de maior número de vértices no grafo de voluntário foi aquela em que $|V| = 23$. Valores maiores que esse exaurem a memória do meu computador. Ainda que o computador tivesse mais memória, seria difícil realizar testes com $|V| > 32$ em máquinas de 32 bits de maneira eficiente (devido a representação máxima de um inteiro não-sinalizado que é calculado para armazenar o tamanho do power set) sem a utilização de bibliotecas/recursos auxiliares.

6. Exercício 6

Como pode ser visto na Figura 3 a curva dada pelo comportamento assintótico cresce mais rapidamente do que aquelas obtidas nos experimentos, mostrados nas Figuras 1 e 2.

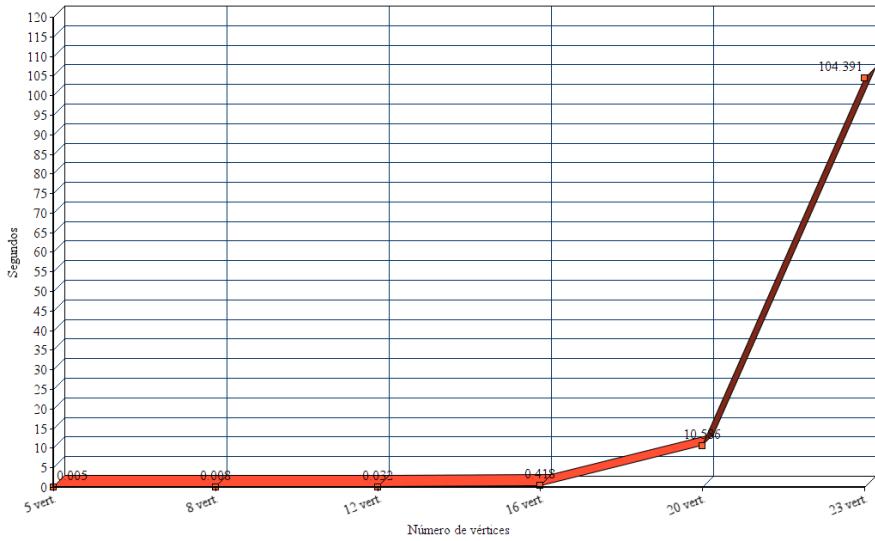


Figure 1. Experimento que o número de focos é aproximadamente metade do número de vértices do grafo original de voluntários.

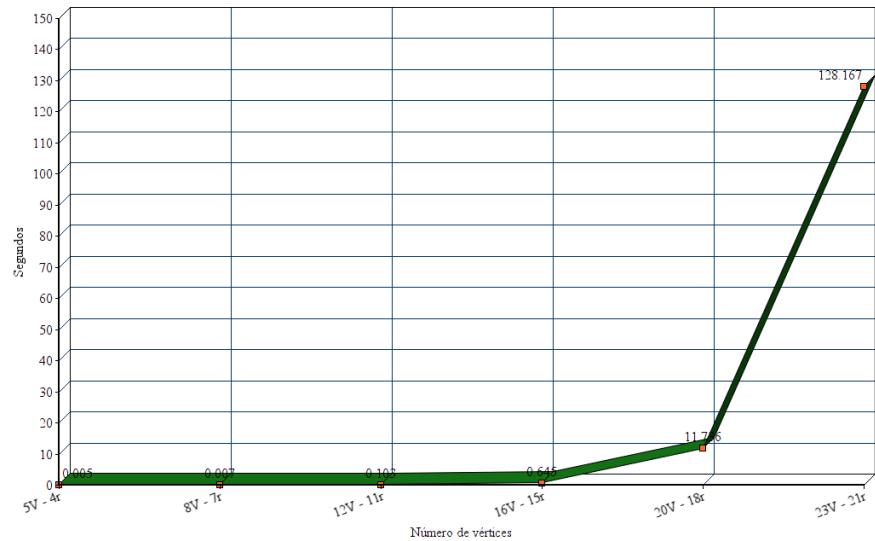


Figure 2. Experimento que o número de focos é aproximadamente igual ao número de vértices do grafo original de voluntários.

Esse comportamento é esperado, pois a análise de complexidade realizada é *pessimista*. Especificamente, ele representa o caso em que todos os subconjuntos do power set correspondente aos vértices do grafo original são calculados; cada um desses power set tem uma incidência de arestas proporcional ao E do grafo original e acessam, individualmente, um número mínimo (ou nenhum) de focos; o teste de conectividade falhará em todos os $2^V - 1$ candidatos, sucedendo apenas no último deles. Ainda assim, apesar dos tempos de execução encontrados nos experimentos serem menores do que aqueles previstos pela análise teórica, o padrão exponencial da curva pode ser visto nas Figuras 1 e 2.

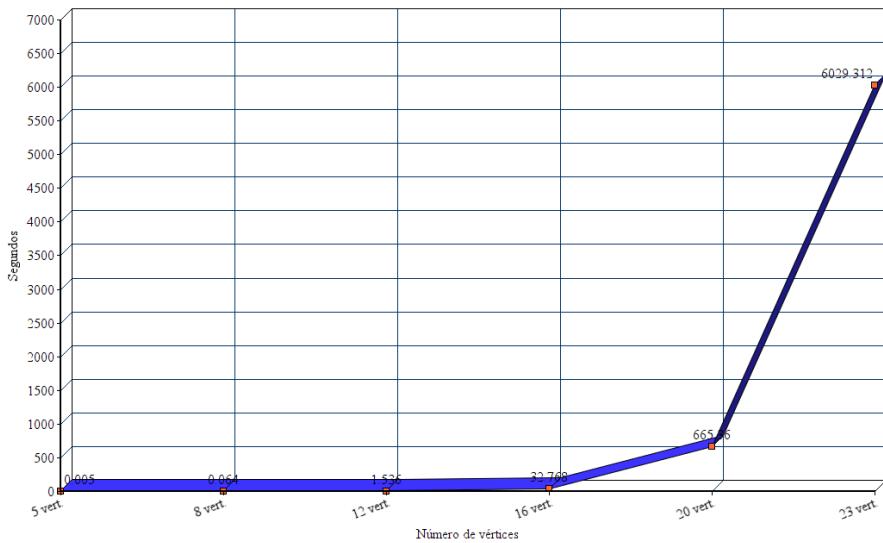


Figure 3. Expectativa teórica máxima baseada apenas no aumento do número de vértices no grafo de voluntários.

References

Trabalho Prático Grafos: ZikaZeroZ

Edson B. de Lima¹

¹Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Belo Horizonte – MG – Brazil

Resumo. Este trabalho descreve a modelagem e implementação de algoritmo em grafos referente a ao processo de descobrimento de vértices, em visitação de caminhos, que contém focos de dengue distribuídos na estrutura de um grafo. O algoritmo utiliza um procedimento modificado de um DFS (Depth First Search) para encontrar o menor conjunto de vértices que contém todos os focos de dengue.

1. General Information

Seja o grafo $G = (V, E)$ com o conjunto de vértices V que representam os vizinhos, e E o conjunto de arestas que conectam os vizinhos. E seja F um conjunto de focos de dengue f_j , com $1 \leq j \leq n$. O problema consiste em encontrar o menor caminho conexo do grafo que contenha todos os focos de dengue f_j . De acordo com as restrições do problema, o grafo G deve então ser um grafo conexo consistir de componentes conexos para percorrer todos os focos $R(v_i)$ com $v_i \in V$ que contenham o total de focos no grafo. O conjunto de focos será distribuído em cada vizinho de tal forma que cada foco deve aparecer pelo menos um vértice v_i de V . Caso haja componentes conexos em G , pelo menos um dos componentes deve conter todos os focos para que seja possível realizar a busca de vizinhança em cada componente de forma independente e obter o resultado final.

2. Modelagem

Em termos de modelagem do problema, o algoritmo deve encontrar a menor árvore de descendentes de um nó $v \in V$ no grafo que contenha todos os focos f_j , utilizando para isso um procedimento de busca. Tal processo de busca deve ser executado para cada vértice em G , assegurando assim que o resultado final contenha as menores distâncias de cada vértice do grafo para todos os outros vértices restantes.

2.1. Estrutura do problema

O grafo G deve ser modelado a partir de uma estrutura de dados que representa uma lista de adjacências contendo uma lista dos vértices vizinhos para cada vértice v_i do grafo G . Cada vértice da lista é representado por uma classe com os atributos, a saber

```
int d;  
int nome;  
Vertice pai;  
List<Aresta> adj;  
List<Integer> focos;
```

Onde cada atributo deve representar respectivamente,

d – número que indica o tempo de abertura de cada vértice. No algoritmo utilizei para controlar qual vértice deveria ser visitado, evitando assim visitar caminho com ciclo.

nome – representa o índice do vértice, sua identificação

pai – representa o vértice pai de outro vértice ao qual se iniciou o processo de visita. Seja a aresta (u, v) no caminho $u \rightarrow v \rightarrow t$, então $pai(v) = u$.

adj – representa a lista de adjacência de um vértice dado, isto é, as arestas incidentes a partir desse vértice.

focos – estrutura que representa a lista de focos de um determinado vértice.

Cada aresta é representada por dois vértices: um vértice de origem e um vértice de destino. O grafo contém, respectivamente, como atributos:

```
List<Vertice> vertices, currentTree, minTree;  
Vector<List<Vertice>> trees;  
List<Aresta> arestas;  
List<Boolean> visitedFocos;  
int minPath = Integer.MAX_VALUE;
```

Uma lista de vértices que representam os vizinhos v_i , uma estrutura de vértices para resgatar os vértices na busca e incluí-los no conjunto de caminhos T que contém todas as árvores em profundidade com todos os focos, uma lista de arestas, a coleção T que deve armazenar todas as árvores mencionadas no percurso e uma estrutura que representa uma a menor lista de vértices que contém todos os focos f_j .

O algoritmo ainda contém um atributo para indicar o tamanho da menor árvore de busca e uma lista de booleanos para que indica se o foco f_j já foi alcançado na busca. Essa lista de booleanos consiste em um atributo do grafo que representa uma lista indicando se cada foco já foi visto ao visitar um vértice na busca. Na modelagem do grafo G, utilizou-se uma lista de adjacências para um grafo direcionado. Assim, para cada aresta (u, v) inserida através da entrada de dados, outra aresta reversa (v, u) também foi inserida no grafo G.

2.2. Arquivos adicionais

Para auxiliar no processo de leitura e gravação de dados, foi utilizada a classe `Arquivos.java`

Esta classe contém métodos para ler os valores do arquivo de entrada e posteriormente escrever a resposta final em um arquivo de saída criado pela própria classe [Macedo 2008]. Alguns métodos utilizados da classe:

`readInt()`:

Método que recupera um valor inteiro presente na posição atual da linha corrente da entrada

`isEndOfLine()`:

Este método verifica se a leitura do arquivo alcançou o final da linha atual.

`print()`:

Este método realiza a gravação dos dados de resposta no novo arquivo de saída criado.

`close()`:

Método que fecha o arquivo criado e finaliza o processo de entrada e saída, gerando por fim o arquivo de saída.

3. Descrição do algoritmo

O algoritmo começa configurando uma nova instância do grafo a partir dos dados fornecidos no arquivo de entrada. O procedimento de busca segue execução a partir de todos os vértices para encontrar o caminho que contém menor número de vértices com todos os focos. A correção do algoritmo consiste em encontrar a partir deste menor caminho, o menor subconjunto que contém os vértices com todos os focos.

A cada iteração no processo de visita para cada nó na busca, o algoritmo verifica se cada foco encontrado foi marcado na lista de booleanos desses focos. Caso o vértice v_i não tenha sido marcado, marca-o na lista e procede a busca enquanto a lista de focos não foi completamente vista. A partir das arestas que conectam os nós vizinhos na lista de adjacência, segue-se o procedimento de busca em profundidade para cada um desses vértices, representados por u .

```
for (int i = 0; i < v.adj.size(); i++) {  
    Vertice u = v.adj.get(i).destino;
```

Após marcar a lista de focos encontrados do vértice atualmente visitado, o algoritmo verifica se todos os focos foram visitados. Caso isso não aconteça, calcula-se o tempo de abertura do vértice atual v e o novo tempo de abertura para cada vértice u na lista de adjacência de v . No processo de visita, cada vértice é incluído em uma nova lista de vértices que representa a árvore na profundidade atual. Caso todos os focos tenham sido vistos, insere-se a árvore na profundidade em uma estrutura que representa o conjunto de todas as árvores na profundidade que contém os vértices com todos os focos alcançados, isto é, o conjunto de caminhos que contém os vértices com todos os focos. A cada iteração desse procedimento, reinicia-se a contagem dos tempos de abertura dos vértices e todos os focos são reiniciados como não visitados.

Tem-se que ao final do processo de visita em todos os vértices, encontramos um conjunto T de caminhos contendo as possibilidades de alcançar todos os focos partindo de cada vértice $v_i \in V$. O algoritmo, então procede com a tarefa de encontrar o menor desses caminhos, o qual deve conter a menor distância a ser encontrada com todos os focos. Nota-se que a menor árvore encontrada contém o menor caminho entre todos os focos como subconjunto. Portanto, o último procedimento a ser realizado consiste em conseguir o menor subconjunto de vértices que contém todos os focos visitados a partir do último vértice na árvore.

Suponha que a árvore encontrada na busca contém os vértices.

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7$

Porém, sabe-se que o menor conjunto contendo todos os focos consistem apenas nos vértices $v_i = 2, 3, 4, 5$. Dessa forma, o algoritmo verifica na lista de vértices da árvore qual o menor subconjunto de vértices com todos os focos f_j remarcando cada foco presente em cada vértice na lista a partir do último vértice. No exemplo acima, o algoritmo encontra todos os focos alcançados desde $v_i = 7$ até $v_j = 2$. O menor subconjunto encontrado, então, consiste no caminho.

$2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7$

O menor subárvore do conjunto de árvores com todos os focos refere-se àquela cuja sequencia se inicia a partir dos primeiros vértices. Isso implica consequentemente

que a soma dos vértices será a menor em termos de empate caso haja mais de uma árvore mínima com mesmo tamanho. A árvore de fato apresenta a menor soma por iniciar dos menores vértices e seguir o processo de visita pela lista de adjacências em ordem crescente de visitação.

Para fugir de armadilhas, o procedimento que visita para todos vértices, analisa antes se o grafo contém vértice sem outros vértices adjacentes, ou seja, se existem vizinhos que não estão colaborando com nenhum outro. Caso isso aconteça, o algoritmo verifica se algum dos vértices isolados contém todos os focos. Nesta situação, o menor subconjunto de vértices consiste apenas do primeiro vértice isolado contendo todos os focos. Ao final do algoritmo, após obter a resposta do menor caminho, o algoritmo utiliza o Quicksort para ordenar a resposta de saída final seguindo a visitação em ordem crescente pela lista de vértices seguindo a lista de adjacências de cada vértice na busca.

4. Complexidade temporal e espacial

A implementação do algoritmo utiliza o processo de busca em profundidade modificado para encontrar o número de focos. Cada vértice do grafo inicia o processo de visitar outros vértices que estão na sua lista de adjacência, de forma que no pior caso a busca alcance todos os vértices do grafo. Como cada aresta deve ser percorrida, tem-se que a busca em profundidade considerando cada vértice do grafo v_i visitado custa $\Theta(V + E)$ [Cormen]. Cada vértice porém deve percorrer sua lista de focos para marcar aqueles que não foram marcados na lista de todos focos do grafo, o que custa $(V - 1)(F - 1) + F$ operações, no caso em que todos os $v_{1,|V|-1}$ vértices contém os $f_{1,|F|-1}$ focos e o último vértice v_n contém todos os f_j focos, com $1 \leq j \leq m$ e $|F| = m$. O processo de visita em um novo vértice verifica se todos os focos já foram alcançados. Reinicia-se a configuração dos vértices como não visitados caso todos já tenham sido vistos, custando assim $\Theta(F)$ operações. Então para cada vértice vértice, o algoritmo realiza $\Theta(VE + VF)$ operações. Por fim, o algoritmo deve custar $\Theta(V^2E + V^2F)$ para a busca em profundidade iniciando a partir de cada vértice $v_i \in V$.

5. Casos de teste

Para a análise de casos de testes da implementação utilizada na linguagem JAVA, um conjunto de configurações de grafos foi utilizado para verificar a corretude e eficiência do algoritmo utilizado.

Nas primeiras versões do algoritmo, a solução ótima não era encontrada devido ao processo de visita reiniciar a contagem dos focos vistos e os tempos de início de cada vértice na árvore em profundidade. Assim, ao encontrar todos os focos em uma árvore na profundidade de um grafo completo, a busca continua das arestas restantes do último vértice no qual a árvore foi encontrada para reiniciar o tempo de contagem dos vértices. Ao retomar a lista de adjacência dos demais vértices na árvore sua lista de adjacência acaba sem processo de busca por conter em todos os demais vértices um tempo de abertura diferente de zero.

Casos de teste com entradas variadas foram realizados em (Figure 1) para analisar o desempenho do algoritmo com diferentes configurações de grafos.

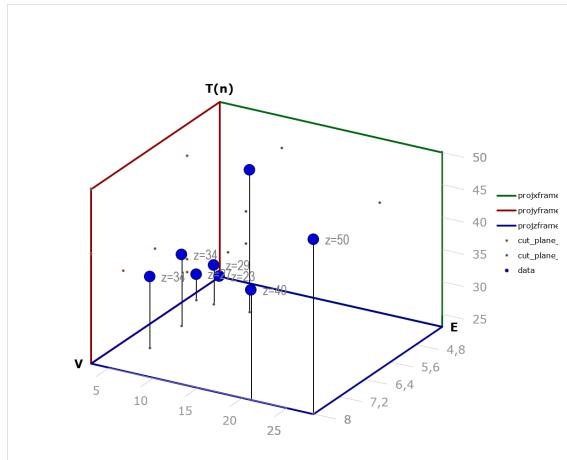


Figura 1. Casos de teste com o conjunto de entradas fornecido

Tabela 1. Tempo de execução do algoritmo ZikaZeroZ com diversas configurações de $G=(V,E)$ e $F = f_1, f_2, \dots, f_n$

Entradas	V	E	F	$T(n)$
Entrada 1	5	6	4	29
Entrada 2	6	6	6	34
Entrada 3	7	6	13	34
Entrada 4	8	28	8	50
Entrada 5	4	3	4	23
Entrada 6	8	28	8	-
Entrada 7	5	4	5	27
Entrada 8	5	10	8	45

6. Análise de resultados

A análise de teste mostrou que o tempo de execução para encontrar a solução ótima é maior para grafos mais densos, principalmente aqueles que contêm maior quantidade de ciclos, incluindo grafos de configuração próximos a um grafo completo de tamanho $|V|$. Tendo em vista esses resultados, a complexidade justifica o maior tempo de execução nesses casos. Como a complexidade do algoritmo depende do número de vértices $|V|$, do número de arestas $|E|$ e do número de focos $|F|$ com $T(n) = \Theta(V^2E + V^2F)$, no pior caso, $|E| = |V|^2$. Assim sendo, $T(n) = \Theta(V^4 + V^2F)$, que deve ser mais custoso considerando um grafo denso que um grafo esparsa.

Referências

- Cormen, T. *Algoritmos Teoria e Prática*. 3th edition.
 Macedo, E. (2008). *Arquivos algoritmos para entrada e saída*.

Projeto e Análise de Algoritmos: Trabalho Prático de Grafos - ZikaZeroZ

Luiz Henrique Santos
luiz.santos@dcc.ufmg.br

Exercício 1

Dados N voluntários e R focos de reprodução, onde cada voluntário tem acesso a um conjunto de focos, deseja-se encontrar o menor número de voluntários que consiga cobrir todos os focos, com a restrição de que cada voluntário deve conhecer ao menos um outro voluntário no grupo final. Colocando os voluntários como vértices e as relações de amizade como arestas, já temos aí um problema modelado em grafos.

Seja $P(R)$ o conjunto potência de R , ou seja, o conjunto que contém todos os subconjuntos de R . Temos que cada voluntário (i.e. vértice) tem um elemento de $P(R)$ associado a ele. Encontrar o menor número de elementos em $P(R)$ cuja união reconstitua integralmente o conjunto original R é um problema clássico NP-Completo, conhecido na literatura como Cobertura de Conjuntos (*Set Cover*).

Para este problema, existe a restrição adicional de que o subgrafo induzido da escolha dos voluntários seja conexo.

Exercício 2

Como o problema é NP-Completo e exige-se a solução ótima, precisamos verificar todos os subgrafos possíveis. Como a maior instância executável em tempo hábil é com $|V| < 32$, utilizamos máscaras de bits do tipo `unsigned` para representar cada subconjunto. No pseudo-código Algorithm 1 mostrado abaixo temos o loop principal que executa força bruta, gerando todas as máscaras válidas. Como geramos todas as máscaras iterativamente, o desempate acontece de forma natural selecionando a sequência de menor ordem lexicográfica, e a corretude fica provada pela natureza da busca exaustiva.

Em Algorithm 2 verificamos se todos os focos de Zika estão sendo cobertos, e o Algorithm 3 mostra como é feita a busca em largura no grafo para testar conectividade.

Algorithm 1: Loop principal do algoritmo de força-bruta

```
input : Grafo de voluntários G
output: Subgrafo induzido ótimo

inicializa optimal_mask com todos os bits setados pra 1;
for subset_mask ← 1 to  $2^{|V|} - 1$  do
    if  $G.coverAllSources(subset\_mask)$  and  $G.isSubgraphConnected(subset\_mask)$  then
        if subset_mask é melhor que optimal_mask then
            atualiza optimal_mask;
        end
    end
end
```

Algorithm 2: Função coverAllSources

```
input : subset_mask, G(V,E)
output: Booleano indicando se a cobertura foi total ou não

cria um conjunto vazio S de inteiros;
for all  $v \in V$  do
    if  $v.maskID$  está marcado em subset_mask then
        | insere os focos conhecidos por v no conjunto;
    end
end

if cardinalidade de S == número de focos then
    | return True;
end
return False;
```

Algorithm 3: Função isSubgraphConnected

input : subset_mask, G(V,E)
output: Booleano indicando se o subgrafo é conexo

marca todos os vértices como não visitados;
seleciona o primeiro vértice (i.e. bit == 1) em subset_mask;
coloca esse vértice numa fila;

while fila não está vazia **do**

- current \leftarrow frente da fila;
- remove frente da fila;
- if** marca como visitado;
- for** all v in $AdjList[current]$ **do**

 - if** não foi visitado e faz parte de subset_mask **then**

 - $v \rightarrow$ final da fila;

 - end**

- end**
- then** current não foi visitado
- end**

- end**

for all v in V **do**

- if** está na máscara subset_mask e não foi visitado **then**

 - return** False;

- end**

- end**

return True;

Exercício 3

Tempo. O loop principal da força bruta executa em $O(2^V)$, pois geramos todas as máscaras possíveis para V vértices. O intuito por trás das máscaras é fazer com que todas as operações que as utilizem sejam executadas em tempo constante. A função coverAllSources analisa, no pior caso, todos os focos R conhecidos por todos os voluntários V, sendo $O(VR)$ no pior caso. A função isSubgraphConnected faz uma simples busca em largura em $O(V + E)$ e tem um passo adicional que verifica em $O(V)$ se todos os vértices da máscara foram visitados, resultando em $O(V + E)$. A complexidade assintótica final do algoritmo fica $O(2^V * (VR + E))$.

Espaço. Foram implementadas duas classes para os algoritmos: o tipo *Node* para os vértices e o tipo *Graph* para o grafo. Cada objeto do tipo Node possui uma lista de adjacência e uma lista de focos do mosquito, ocupando, no pior caso, um total de $O(E + R)$ em memória. O tipo Graph possui apenas uma lista do tipo Node, o que para V vértices resulta numa complexidade final de memória de $O(V * (E + R))$.

Exercício 5

Os gráficos abaixo foram gerados a partir da média aritmética de 10 execuções em ambiente Linux, num computador do DCC com 4GB de memória e processador Intel Core 2 Duo CPU E8400 (3.00GHz). O gráfico 1 engloba todos os dados, onde a máxima instância executada possuía 27 vértices. O gráfico 2 foi gerado a partir das mesmas informações, sendo simplesmente um recorte dos dados para execuções até 20 vértices. Todos os grafos foram grafos completos com R se aproximando da metade de N.

Foram feitas também instâncias de caráter mais específico, como um caminho simples de N vértices, onde o primeiro vértice tinha um conjunto de focos A, o último vértice do caminho tinha um conjunto de focos B, e os demais vértices conheciam um conjunto de focos C, com A, B e C completamente disjuntos. Para obter uma solução ótima nessa caso somos forçados a selecionar todos os vértices para atender às restrições do problema.

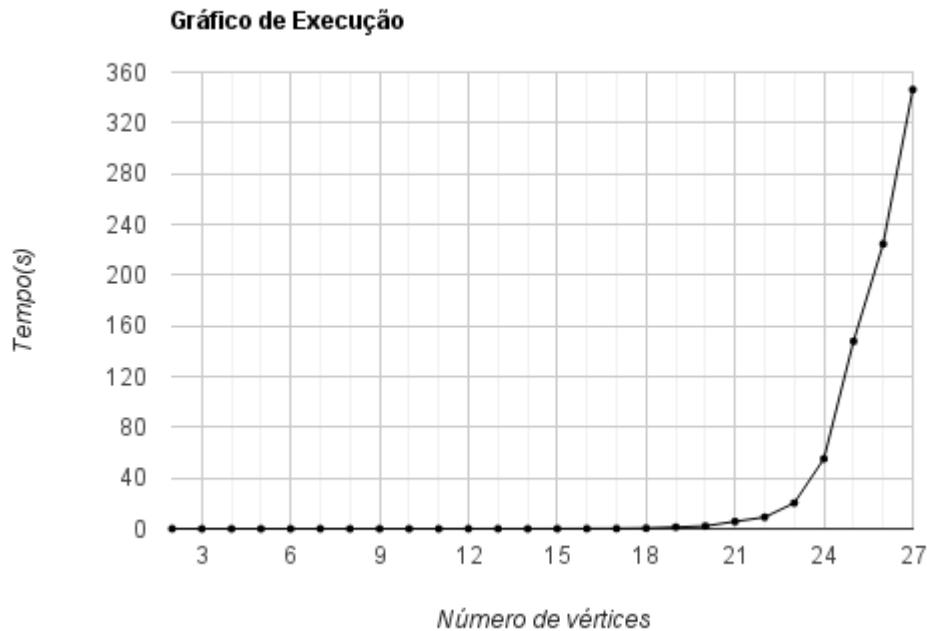


Figure 1: Curva de crescimento para N até 27 num grafo K_N com $R \approx \frac{N}{2}$

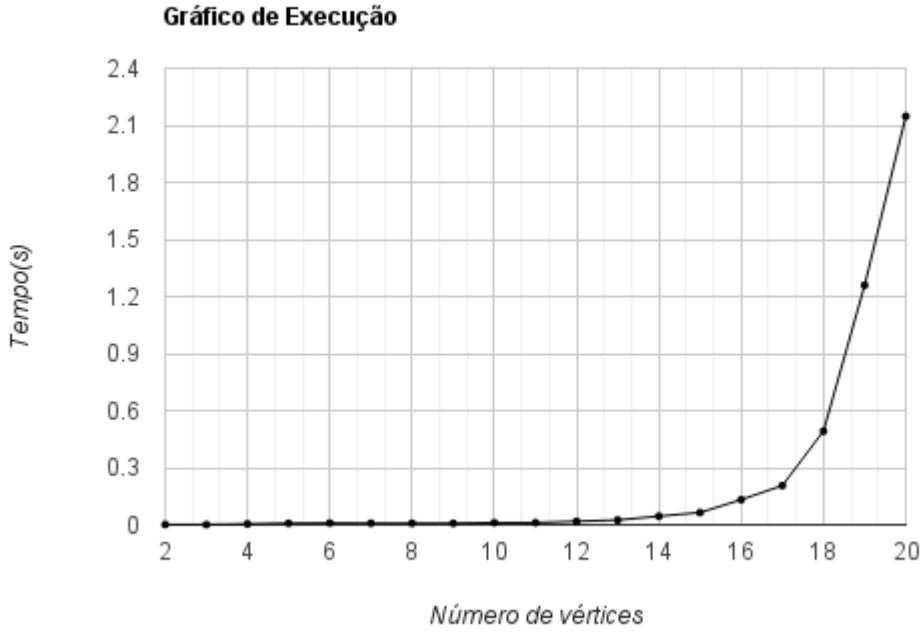


Figure 2: Curva de crescimento para N até 16 num grafo K_N com $R \approx \frac{N}{2}$

Exercício 6

Os experimentos feitos revelam que a curva de crescimento do tempo é exponencial, como era esperado pela complexidade assintótica encontrada no Exercício 3. No primeiro gráfico percebemos que a curva é muito achatada até $N = 20$. Quando fazemos um recorte da mesma execução com N até 16, percebemos que ainda assim o caráter exponencial se mantém.

Projeto e Análise de Algoritmos

Trabalho Prático 1

Mariana Arantes

¹Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

{marianna.arantes}@dcc.ufmg.br

Exercício 1

O surto da doença *Zika*, causada pelo *Zika* vírus, tem assustado a população brasileira. Nos últimos meses diversos pacientes foram diagnosticados com a doença e passaram por sérias complicações. Devido à possibilidade do avanço da doença para todos os estados do Brasil e para outros países da América Latina, diversas campanhas estão sendo realizadas para combater o mosquito *Aedes aegypti*, transmissor do vírus.

Uma forma moderna de combater o mosquito se encontra no uso das redes sociais virtuais. Cada usuário da rede tem acesso a locais específicos que podem conter focos do mosquito. Esses usuários podem trabalhar como voluntários acessando os focos próximos a eles. Para realizar o extermínio dos mosquitos da melhor forma possível, basta selecionar na rede um conjunto de amigos dispostos a ajudar de forma que todos os focos do mosquito sejam alcançados e o número de voluntários seja mínimo.

Esse problema pode ser modelado através do uso de grafos. Os vértices representam os voluntários, as arestas representam as relações de amizade entre eles e os focos aos quais cada voluntário tem acesso são representados como atributos do vértice. Segue a definição formal.

Dados **um grafo** não direcionado $G(V, E)$ onde V é o conjunto de n voluntários e E é o conjunto de m relações de amizade, **um conjunto** \mathbb{F} de r elementos, $r \in \mathbb{N}$, que representam os focos do mosquito, **uma relação** $\mathbb{R}(v) : V \rightarrow \mathbb{F}$ definida para cada $v \in V$ e **uma função** $g : V' \rightarrow \mathbb{N}$, onde $V' \subseteq V$, tal que $g(V') = |\bigcup_{v \in V} \mathbb{R}(v)|$. O objetivo é selecionar o menor subgrafo conexo de V , chamado de V' , que satisfaça $g(V') = r$.

Algumas suposições são feitas para a solução do problema: (i) o grafo de entrada sempre vai possuir solução, ou seja, sempre existe um subgrafo conexo que cobre todos os focos, (ii) todo vértice possui pelo menos um foco do mosquito (o arquivo de entrada não possui linhas vazias) e (iii) o número de focos é no mínimo um.

A Figura 1 mostra um exemplo de modelagem para um grafo com 3 vértices, 3 arestas e 3 focos do mosquito. O menor subgrafo conexo que cobre todos os vértices é o subgrafo induzido pelos vértices 1 e 3, representados em azul na figura.

Exercício 2

O algoritmo proposto para resolver o problema verifica todos os subconjuntos possíveis de V . Para cada subconjunto V' , o valor da função $g(V')$, definida no exercício 1, é calculado e a condição de cobertura de todos os focos é verificada. Se o subconjunto

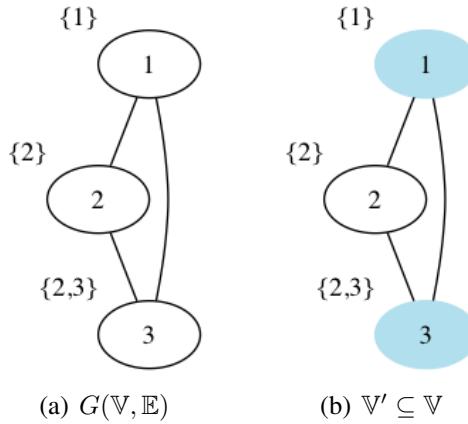


Figura 1. Exemplo de modelagem usando grafo. A figura (a) mostra o grafo de entrada e a figura (b) mostra a solução para o problema.

cobrir todos os focos, o subgrafo de G induzido por V' é criado. Se esse subgrafo induzido for conexo, a solução para o problema foi encontrada e o algoritmo encerra sua execução. Para garantir que a primeira solução encontrada seja a ótima, ou seja, a que contém o menor número de vértices possível, os subconjuntos são verificados em ordem crescente de tamanho e os vértices são ordenados lexicograficamente.

O método ZikaZeroZ é mostrado no Algoritmo 1. Os dois laços externos são responsáveis por percorrer todos os subconjuntos do conjunto V . A variável *subsetSize* contém o tamanho do subconjunto a ser gerado e a variável V' contém o conjunto de vértices que pertencem ao subconjunto. Para cada subconjunto, todos os vértices pertencentes a ele são percorridos para verificar se todos os focos são cobertos e, caso eles sejam, um subgrafo induzido por V' é criado e o algoritmo de busca em largura é executado para um vértice de origem aleatório. Se o algoritmo de busca em largura visitar todos os vértices, o grafo é conexo e a solução foi encontrada. Se o subgrafo não for conexo ou não cobrir todos os focos, ele é desconsiderado e a busca continua.

O subgrafo induzido por V' é construído com o Algoritmo 2. Para todo vértice $u \in V'$, sua lista de adjacência no grafo original G é percorrida e para cada vértice v adjacente a u , se $v \in V'$, uma aresta (u,v) é adicionada ao subgrafo induzido G' .

O método de busca em largura (BFS) é mostrado no Algoritmo 3. A implementação desse algoritmo foi realizada de forma simplificada, pois a única informação necessária para o problema é a dos vértices que foram visitados durante a execução. Essa informação é mantida no vetor *visited* que contém 0 caso o vértice não tenha sido visitado e 1 caso o vértice tenha sido visitado. O Algoritmo 1 verifica se o subgrafo é conexo fazendo uma comparação entre a soma dos valores do vetor *visited* e o número de vértices do subgrafo.

Os três algoritmos fazem uso da implementação do grafo por meio de uma lista de adjacências. Os focos alcançáveis por cada vértice são representados por um conjunto e armazenados como um atributo do vértice.

```

Input: A graph  $G(V, E)$ , a positive integer  $r$  and a relation  $\mathbb{R}(v) : \mathbb{V} \rightarrow \mathbb{F}$ 
Output: A subset  $V' \subseteq V$ 
for  $subsetSize \leftarrow 1$  to  $|G.V|$  do
    for each  $V'$  in  $C_{|G.V|, subsetSize}$  in lexicographic order do
        // check if subset covers all spots
         $spots \leftarrow set()$ 
        for each  $v \in V'$  do
            |  $spots \leftarrow spots \cup R(v)$ 
        end
        // check if induced subgraph is connected
        if  $spots.size == r$  then
            |  $subgraph \leftarrow$  induced subgraph of  $G$  by  $V'$ 
            |  $source \leftarrow subgraph.source()$ 
            |  $visited \leftarrow subgraph.BFS(source)$ 
            | if  $visited.sum() == subsetSize$  then
                | | return  $V'$ 
            | end
        end
    end
end

```

Algorithm 1: ZIKAZEROZ

```

Input: A graph  $G(V, E)$  and a subset  $V' \subseteq V$ 
Output: An induced subgraph  $G'(V', E')$  of  $G$ 
Let  $G'$  be a new graph with  $|V'|$  vertices
for each vertex  $u \in G.V$  do
    if  $u$  is in  $V'$  then
        // search adjacency list of u
        for each vertex  $v \in G.adj[u]$  do
            if  $v$  is in  $V'$  then
                | // edge is still an edge of  $G'$ 
                |  $G'.add\_edge(u, v)$ 
            end
        end
    end
return  $G'$ 

```

Algorithm 2: INDUCED SUBGRAPH

Exercício 3

O algoritmo ZikaZeroZ proposto encontra a solução através da busca por todos os subconjuntos do conjunto V . Para cada subconjunto encontrado, o algoritmo realiza uma verificação em três etapas para saber se o subconjunto é a solução. Primeiro, o número de focos cobertos pelos vértices do subconjunto é verificado, em seguida o subgrafo induzido é criado e por último uma busca em largura é realizada.

O melhor caso acontece quando o primeiro subconjunto a ser testado (subconjunto

```

Input: A graph  $G(V, E)$ 
Output: An array of size  $|G.V|$ 
for each vertex  $u \in G.V - \{s\}$  do
    |  $visited[u] \leftarrow 0$ 
end
 $visited[s] \leftarrow 1$ 
 $Q = \emptyset$ 
enqueue( $Q, s$ )
while  $Q \neq \emptyset$  do
    |  $u = dequeue(Q)$ 
    | for each vertex  $v \in G.adj[u]$  do
        |   | if  $visited[v] == 0$  then
        |   |   |  $visited[v] \leftarrow 1$ 
        |   |   | enqueue( $Q, v$ )
        |   | end
    | end
end
return  $visited$ 

```

Algorithm 3: BREADTH FIRST SEARCH (BFS)

que contém apenas o vértice 1) é a solução para o problema. Nesse caso, a etapa de realizar a cobertura dos focos é realizada em $O(1)$, pois é uma busca linear em um único vértice, o subgrafo induzido é calculado em $O(1)$, pois um subgrafo induzido com apenas um vértice não possui arestas e a busca em largura também é realizada em $O(1)$. Logo, no melhor caso, o algoritmo possui complexidade de tempo igual a $O(1)$.

O pior caso acontece quando o último subconjunto a ser testado é a solução para o problema ($V' = V$) e todas as etapas de verificação são realizadas para todos os subconjuntos. Nesse caso, o algoritmo vai executar 2^V iterações e em cada uma, uma busca linear nos vértices do subconjunto vai ser realizada, o que possui complexidade $O(V')$, um subgrafo induzido vai ser construído, o que possui complexidade $O(V + E)$ porque a lista de adjacência do grafo original precisa ser percorrida e por fim, uma busca em largura vai ser realizada, com complexidade $O(V' + E')$. A complexidade final do algoritmo é $O(2^V * ((V') + (V + E) + (V' + E')))$. Como $V' \subseteq V$, a complexidade de tempo é $O(2^V * (V + E))$.

O algoritmo faz uso de uma lista de adjacência para armazenar o grafo, que ocupa espaço $O(V + E)$ e a cada iteração do laço externo (um tamanho $subsetSize$), armazena todas as combinações possíveis de V com tamanho $subsetSize$. Logo a complexidade de espaço é dominada pelo espaço necessário para armazenar as combinações a cada iteração. A iteração que mais ocupa espaço consiste da iteração onde $subsetSize = V/2$, logo a complexidade de espaço é $O(C_{V,V/2})$.

Exercício 5

Para a realização dos testes, um gerador de grafos foi construído. O gerador recebe como parâmetro o número de vértices n , ($n \geq 1$), o número de arestas m , ($(n - 1) \leq m \leq n(n - 1) \div 2$) e o número de focos r , ($r \geq 1$) e produz um grafo conexo com os

parâmetros especificados.

O primeiro passo do gerador consiste em construir uma árvore geradora, para garantir que o grafo gerado é conexo (o problema não exige que o grafo G seja conexo, porém os testes foram executados somente para grafos conexos para facilitar a análise). A árvore geradora é construída com um algoritmo similar ao algoritmo de Prim, que encontra árvores geradoras mínimas. O algoritmo se inicia com um conjunto S vazio e um conjunto V que contém todos os vértices e a cada etapa, escolhe aleatoriamente um vértice u de S e um vértice v de V , adiciona a aresta (u, v) ao grafo, remove v de V e adiciona v em S . A árvore geradora é concluída quando $V = \emptyset$. Após a geração da árvore, arestas aleatórias são inseridas no grafo até que o número de arestas m seja atingido.

Após a escolha das arestas, o gerador de grafos define os focos que são cobertos por cada vértice. A escolha do número de focos cobertos por cada vértice afeta diretamente a análise experimental, pois se poucos vértices possuem muitos focos, a solução para o problema será um subconjunto muito pequeno de V , fazendo com que o algoritmo ZikaZeroZ encontre a solução rapidamente. Após alguns testes, foi definido que o número médio de focos que cada vértice cobre, de forma a melhorar a análise experimental, é igual ao número de focos dividido pelo número de vértices, quando $r \geq n$ e é igual a 1, caso contrário. Os focos são escolhidos de forma aleatória para cada vértice e o gerador garante que todos os vértices possuem pelo menos um foco e que todos os focos são associados a pelo menos um vértice. Essa escolha tenta refletir um cenário real, onde voluntários tendem a acessar o mesmo número de focos cada um, de forma a não sobrecarregar ninguém.

Foram gerados grafos para um número de focos (r) igual a 20 e para o número de vértices (n) variando de 1 a 28. Para cada número de vértices, foi gerado um grafo com $n - 1$ arestas e outro com $\frac{n(n-1)}{2}$ arestas, ou seja, uma árvore e um grafo completo. Esses grafos foram usados como entrada para o algoritmo ZikaZeroZ (Algoritmo 1) e o tempo de execução foi utilizado para a realização das análises. Para cada grafo de entrada, o algoritmo foi executado 3 vezes e as análises mostram a média do tempo das 3 execuções e o desvio padrão.

A Figura 2 mostra o tempo médio de execução e o desvio padrão para os grafos de entrada. O tempo de execução é mostrado em azul para grafos completos e em vermelho para árvores. É possível perceber que o algoritmo encontra a solução em menos tempo para grafos completos. Esse resultado é esperado, pois se o grafo é completo, qualquer subconjunto dele é conexo, aumentando as chances de um subconjunto pequeno satisfazer as condições. Já em um árvore, o número de subconjuntos conexos é pequeno, fazendo com que o algoritmo tenha que verificar muitos subconjuntos, demorando mais. Em ambos os casos, o tempo execução segue um crescimento exponencial em função do número de vértices do grafo de entrada.

Exercício 6

A análise teórica apresentada no exercício 3 nos diz que a complexidade de tempo do algoritmo ZikaZeroZ (Algoritmo 1) é da ordem de 2^V , ou seja, o algoritmo possui complexidade exponencial. Na análise experimental mostrada no exercício 5, foi possível confirmar esse comportamento do algoritmo, pois o tempo médio de execução aumentou

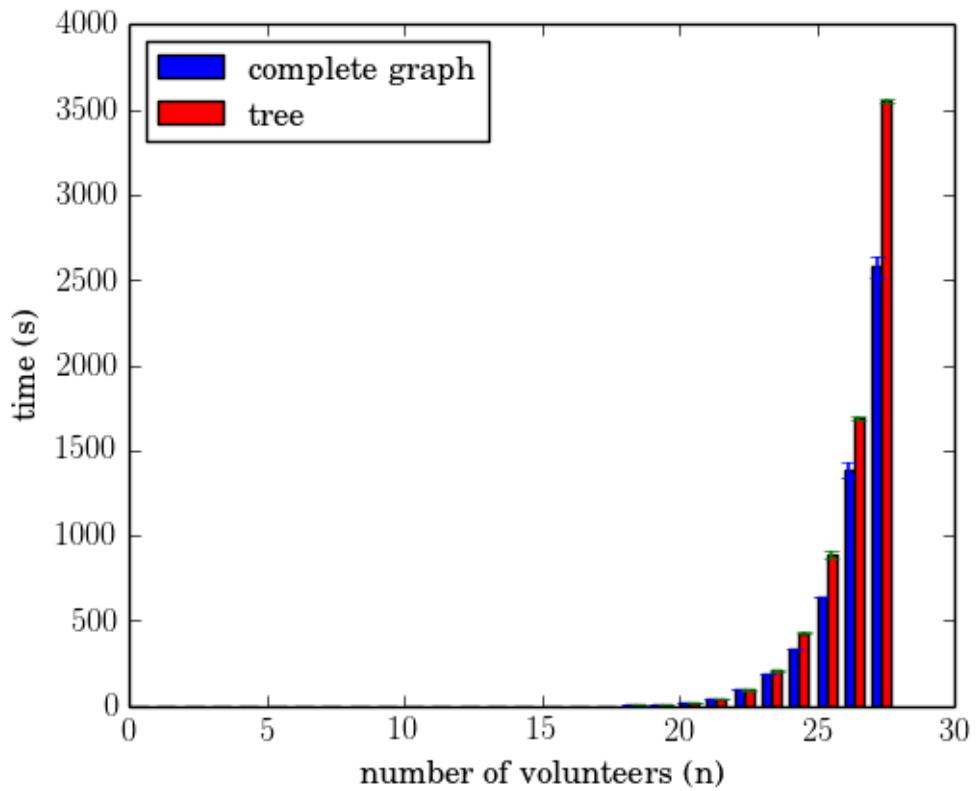


Figura 2. Tempo médio de execução em função do número de voluntários.

de forma exponencial com o aumento do número de vértices.

Trabalho Prático: ZikaZeroZ

Lucas Miguel S. Ponce

¹ Projeto e Análise de Algoritmos

Programa de Pós-Graduação em Ciência da Computação (PPGCC)
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

lucasmsp@gmail.com

Resumo. A principal ação de combate ao mosquito *Aedes aegypti* é evitar a sua reprodução. Um dos possíveis modos de estimular esse combate é se aproveitando das redes sociais virtuais. Esse trabalho propõe uma solução para encontrar o menor grupo de voluntários que abrangem todas as áreas de focos. A solução proposta é um algoritmo baseado na força bruta, a partir de uma modelagem do problema em grafos, analisa para cada pessoa do sistema, o menor conjunto de elementos vizinhos que satisfaça a máxima do problema, isto é, abranger todos os focos com o menor número de pessoas. Os testes realizados obtiveram resultados que corroboram com a modelagem do problema.

1. Exercício 1 – Apresente uma modelagem para o Problema ZikaZeroZ empregando grafos. Deixe claro quais as restrições e/ou suposições devem ser feitas.

O grafo a ser modelado já é definido por um conjunto V que representa os n voluntários, um conjunto A que representa as m amizades entre os voluntários, um conjunto F dos r focos de reprodução do mosquito, e, uma relação $R(v) : V \rightarrow F$, definida para cada $v \in V$, explicitando os focos de reprodução aos quais o voluntário v tem acesso. A solução proposta utiliza a mesma modelagem para o grafo.

Como a premissa do problema é utilizar as redes sociais para formar uma rede (mínima) de voluntários coesa, não haveria sentido em colocar relações unidirecionais, muito menos criar mais de duas relações (i.e., amizade) para um mesmo par de amigos. Também não haveria problemas de distância entre os voluntariados, uma vez que esse utilizam redes sociais para se comunicarem. Diante disso, foi definido que a modelagem seria em um grafo simples, não direcionado e não ponderado.

O fenômeno de redes sociais tem geralmente a característica de formarem grafos esparsos, como exemplos as redes de citações e de amizades [Newman 2003]. Logo, modelar o grafo em uma lista de adjacência seria a estratégia mais condizente com o problema dado.

A implementação da lista de adjacência, será feita em linguagem Java, e será em um vetor de $|V|$ posições de objetos *ListaAdjacencia*. Cada uma posição representando a lista de adjacências de um vértice. Esse objeto contém: um inteiro (podendo, para implementações futuras, ser mudadas para caracteres), uma lista dinâmica de focos (*ArrayList*) e uma lista dinâmica de vértices adjacentes (*ArrayList* de inteiros). A lista completa possui $O(V + E + VF)$ de complexidade espacial. Como falamos de um grafo não direcionado, o tamanho de arestas será o dobro da quantidade caso fosse um grafo direcionado.

Necessariamente, para encontrar a solução desse problema, tem-se q percorrer todos os vértices antes de dar uma solução final, uma vez que poderá sempre existir um vértice que contenha mais pontos de focos (i.e., pode existir uma pessoa ainda não analisada que tem acesso a mais focos ou até mesmo todos os focos).

Sabe-se que para um determinado grafo G não orientado, ao se realizar uma busca em largura, haverá mais de uma possível árvore T e com alturas (níveis) diferentes. A árvore formada dependerá do vértice inicial e da ordem em que se armazenam os vértices adjacentes. A Figura 1 exemplifica duas árvores T geradas a partir do mesmo grafo (à esquerda), uma com vértice inicial V_1 (no centro) e outra com um vértice inicial V_2 (à direita).

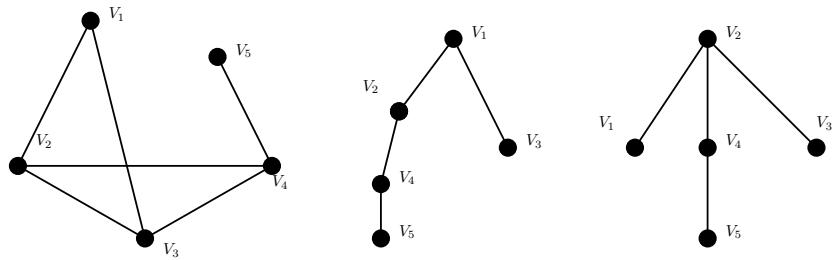


Figure 1. Exemplo de como uma busca em largura pode gerar mais de uma árvore

Sabe-se também que se a árvore é executava n vezes, pelo menos uma vez, o vértice inicial será um vértice que faz parte de V' (vértices da solução ótima). Se o conjunto V' contém dois vértices, então, serão criadas duas vezes uma árvore que o vértice inicial faz parte de V' , e assim por diante.

A abordagem escolhida parte dá ideia de produzir V árvores por meio de uma busca em largura. A criação de cada árvore termina no momento que a altura k possuem vértices com todo o conjunto de focos. Após essa criação, são feitos sucessivos cortes em vértices sem nenhum filho (i.e., são folhas) que são dispensáveis para a solução.

Para tal arbodagem, utiliza-se um vetor de inteiros *count_focos* com um tamanho igual a quantidade distintas de focos (r) para armazenar a quantidade vértices que possuem o foco i . Por exemplo, se *count_focos*[2]= 3, significa que existem três vértices que contém o foco F_2 . E como dito anteriormente, ao utilizar a busca em largura, a cada vértice percorrido, analisa os focos que esse vértice possui e atualiza a quantidade informada pelo *count_focos*. No momento em que todas as posições do *count_focos* forem diferentes de zero, a busca em largura termina. Após isso, realiza uma etapa de corte de vértices (que são folhas da árvore formada) que não são imprescindíveis. Um vértice é imprescindível para a solução encontrada caso um foco só é encontrado nele.

Esse corte é feito da seguinte maneira: Para todos os vértices v do último nível da árvore, analisa se v é imprescindível, caso não seja, delete-o. Caso tenha deletado o vértice v , se o vértice u não tiver mais nenhum filho, ele se torna a nova folha para esse galho da árvore (i.e., muda o nível de u para ser uma folha). Esse processo continua para todas as outras folhas não vistas até não ser possível realizar algum corte.

Ao terminar o corte, algoritmo armazena a solução encontrada (caso tenha sido

feita apenas uma execução da busca em largura) e invoca uma nova busca em largura a partir de outro vértice. Caso já exista uma solução anteriormente armazenada, é escolhido se a mantém ou troca pela solução recente. Essa troca é feita a partir dos seguintes critérios:

1. O menor conjunto de vértices tem prioridade;
2. Em caso de empate no número de elementos, o conjunto com a menor soma dos índices dos seus voluntários tem preferência;

A Figura 2 ilustra parte do processo do algoritmo. O primeiro grafo é o original, o segundo grafo (uma árvore) é quando ele encontra todos os vértices que satisfazem todos os focos, por fim, o último grafo é a árvore formada com os vértices escolhidos após a retirada dos vértices dispensáveis para essa instância.

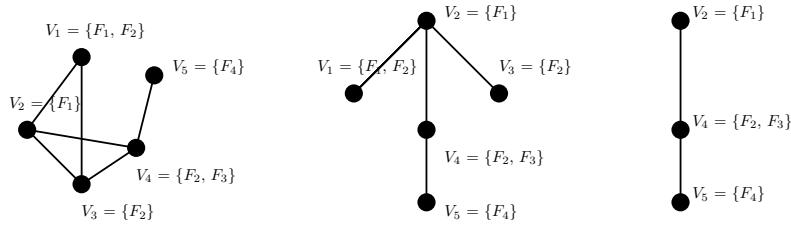


Figure 2. Exemplo da idéia geral do algoritmo

Todo esse processo (i.e., busca em largura, corte de vértices dispensáveis, escolha/armazenamento da melhor solução) é feita para todos os vértices do grafo de entrada.

Como já foi dito, o grafo de entrada será considerado como um grafo não orientado e suas arestas não terão peso. Por motivos de implementação para ganhar performance, tanto os vértices quanto os focos, tem que serem representados por inteiros de 1 à $|V|$ (para os vértices) ou 1 à $|F|$ (para os focos).

Tanto pela ideia por trás de uma pessoa ser voluntária, quanto pela interpretação do problema proposto, foi definido que cada vértice tem que estar relacionado à no mínimo um foco.

2. Exercício 2 – Descreva um algoritmo para resolver o Problema ZikaZeroZ, tendo em vista a modelagem realizada no Exercício 1.

Para facilitar o entendimento do algoritmo, este foi dividido em duas funções, a principal *CoberturaDeFocos_solution()* e a auxiliar *find_group()*. O Algoritmo 1 é um pseudo-código da função principal, ela é a que será chamada e a que retornará o resultado (conjunto de vértices escolhidos). A função basicamente solicita que a função *find_group()* retorne um conjunto de vértices mínimos para cada um dos vértices iniciais $v \in V$. Após obter o retorno desses conjuntos, a função principal compara com a solução anteriormente armazenada e escolhe qual manterá com base nos critérios de decisão anteriormente mencionadas. Após ter definido o melhor conjunto, o algoritmo ordena os vértices por ordem de índices.

O Algoritmo 2 é um pseudo-código da função *find_group()* responsável por realizar a busca em largura a partir de um vértice informado pela função *CoberturaDeFocos_solution()* até chegar à uma altura k onde todos os focos foram preenchidos por pelo

Data: $G = (V, A)$
Result: V' tal que $V' \subseteq V$

```

begin
     $V' \leftarrow \emptyset;$ 
     $X \leftarrow \emptyset;$ 
    for  $v \in V$  do
         $X \leftarrow find\_group(v);$ 
        if  $|X| < |V'|$  then
             $|V' \leftarrow X;$ 
        end
        else if  $|X| == |V'|$  then
             $|V' \leftarrow conjunto com menor soma dos valores dos índices entre V' e X;$ 
        end
    end
    ordena  $V';$ 
end

```

Algoritmo 1: *CoberturaDeFocos_solution()* - Função Principal

menos um vértice da árvore formada. De posse da árvore formada, a função corta os vértices que podem ser removidos da árvore (i.e., vértices dispensáveis). Para decidir se o vértice é dispensável, o algoritmo analisa se ele é um vértice com altura k e se os focos que ele atua não é único para ele. Caso possa ser deletado, o algoritmo analisa o vértice pai p tem outro filho para que possa ser decidido se aumenta a altura de p (para deixar-lo sendo uma folha explicitamente). O algoritmo também procura deletar vértices folhas de galhos menores (com altura menor que k), para isso ele tem que analisar se o vértice não tem filhos (percorrendo o vetor *antecessor*[J]) e se não possui focos únicos para ele.

Esse processo é feito até que se percorra a árvore completamente e não se tenha feito mais nenhum corte. Caso contrário, verifica se com a atualização feita é possível remover um outro vértice.

3. Exercício 3 – Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) do algoritmo proposto no Exercício 2.

A complexidade Espacial da função *CoberturaDeFocos_solution()* dependerá de:

- tamanho do grafo de G entrada – $O(V + E + VF)$;
- tamanho do conjunto solução encontrada – $O(V)$;
- tamanho do conjunto solução encontrada anteriormente – $O(V)$;
- tamanho gasto pela função *find_group()*;

Já a complexidade Espacial da função *find_group()* dependerá de:

- tamanho do grafo de entrada G – $O(V + E + VF)$;
- tamanho do conjunto de vértices encontrado *v_induced*[] – $O(V)$?;
- tamanho do vetor *nivel*[] – $O(V)$;
- tamanho do vetor *antecessor*[] – $O(V)$;
- tamanho do vetor *focos_induced*[] – $O(F)$;
- tamanho da lista fifo *Q* – $O(V)$;

Data: $G = (V, A)$, F e um vértice inicial u

Result: $v_{induced}$ tal que $v_{induced} \subseteq V$

begin

```
// Inicializa as variáveis;  
vinduced ← ∅;  
pred[1...V] ← ∅;  
focosinduced[1...F] ← ∅;  
nível[1...V] ← ∅;  
 $Q \leftarrow u$ ; // uma fila fifo;  
ncompleto ← true;  
vinduced ← ∅;  
//BFS ;  
while ( $Q \neq \emptyset$ ) e (ncompleto) do  
     $v \leftarrow$  primeiro elemento da fifo  $Q$ ;  
    for para cada vértice novo  $w$  que seja filho de  $v$  do  
        adiciona  $w$  em  $Q$ ;;  
        antecessor[w] ←  $v$ ;;  
        nível[w] ← altura da árvore;;  
        for para cada foco  $i$  que  $w$  possua do  
            | focosinduced[i] ← focosinduced[i] + 1;  
        end  
        for  $j \leftarrow 1 \dots |F|$  do  
            | if focosinduced[j] == 0 then  
            |     | ncompleto = false;;  
            | end  
        end  
    end  
    end  
// Corte das folhas ;  
do  
    for para cada  $v \subseteq v_{induced}$  do  
        if  $v$  é uma folha then  
            | if  $v$  tem algum foco que só ele atua then  
            |     | dispensavel ← false;;  
            | end  
            | else  
            |     | dispensavel ← true;;  
            | end  
            | if dispensavel then  
            |     | exclui  $v$  da lista de  $v_{induced}$ ;  
            |     | faz o seu vértice  $w$  antecessor se tornar a nova folha (caso  $w$  não tenha mais nenhum filho);;  
            |     | remove a participação de  $v$  no vetor focosinduced[];  
            | end  
        end  
    end  
end  
while um vértice foi cortado;
```

end

Algoritmo 2: find_group() - Função para encontrar um grupo de vértices

- tamanho das listas auxiliares e temporárias com os focos de um vértice – $O(V)$;

Mesmo que uma implementação em Java exija que valores passados por parâmetro sejam duplicados, isso não interfere na solução da análise, pois além de podermos escrever o grafo G como uma “variável global”, como estamos analisando em notação de assintótica, o valor a ser encontrado é um limite superior e proporcional. Com base nisso, para algoritmo como um todo (i.e., primeira e segunda função), a ordem de complexidade Espacial será dada pela complexidade das duas funções juntas, ou seja:

$$O((V + E + VF + V + V) + (V + V + V + F + V + V)) \quad (1)$$

Como tanto V , quanto E e F são inteiros maiores que 1, temos:

$$= O(E + VF) \quad (2)$$

A complexidade temporal, considerando as operações, da função *CoberturaDeFocos_solution()* dependerá principalmente de $|V|$ vezes a execução da função *find_group()* mais $|V|$ vezes a escolha de qual conjunto de vértices irá ser mantida e por fim, uma ordenação utilizando uma adaptação do algoritmo *mergesort* pela classe *Java.util* que garante uma complexidade de $O(V \log V)$ do conjunto final.

Já a complexidade temporal, considerando as operações, da função *find_group()* dependerá de:

- Inicialização e configuração dos vetores e variáveis temporárias – $O(V)$;
- Execução da BFS, sendo que a cada vértice extraído, precisa conferir se todos os focos já foram preenchidos, ou seja, no pior caso, para cada vértice será percorrido o vetor *focos_induced[]* completamente – $O((V + E) * F)$;
- Para a parte de corte de folhas, a análise será mais detalhada: O pior caso nessa parte seria retirar cada elemento de uma árvore a cada laço interno por vez (i.e., eliminando um galho da árvore). Para decidir se retira ou não um vértice, ele tem que estar na folha (ter altura máxima) ou não ser um vértice inicial e não ter vértices filhos. Esse processo acaba quando após percorrer todos os vértices da árvore e não ser feita mais nenhuma mudança. Para verificar se o vértice tem filhos gasta-se $O(V)$, para verificar se o vértice pai tem outro filho (para decidir se aumenta a altura) gasta-se $O(V)$ e por fim, gasta-se $O(F)$ para avaliar se o vértice tem focos que são importantes. Esse processo é feito até eliminar todos os vértices possíveis da árvore ou seja $O(V)$. Totalizando então para essa parte: $O(((V + E) * F + V + V + V) * V) = O(V^2 F + VEF + V^2)$.

Com posse das análises das duas funções, podemos agora calcular a complexidade temporal total do algoritmo proposto:

$$O(V \log V + V * ((V^2 F + VEF + V^2) + (V + E) * F + V)) \quad (3)$$

$$= O(V \log V + V^3 F + V^3 + V^2 EF + V^2 + V^E F + VEF) \quad (4)$$

$$= O(V^3 F + V^2 EF) \quad (5)$$

Para grafos densos (i.e., $E \approx V^2$):

$$O(V^4F) \quad (6)$$

Embora se tenha achado uma complexidade assintótica de $O(V^3F + V^2EF)$, na verdade, será maior que isso, uma vez que não foi considerado algumas complexidades de baixo nível, como por exemplo a complexidade de remover um índice em um *ArrayList*.

4. Exercício 4 – Implemente o algoritmo proposto no Exercício 2 na linguagem de programação C, C++, Java ou Python.

Implementação feita em linguagem Java, as três classes relevantes ao problema se encontram no apêndice deste relatório. O código fonte completo se encontra na própria pasta junto ao relatório.

5. Exercício 5 – Execute testes da implementação do Exercício 4, propondo instâncias interessantes para o Problema ZikaZeroZ. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n , m e r). Outra sugestão é relatar o tamanho da maior instância para o qual foi produzida uma solução.

Os testes a seguir foram feitos em um computador Desktop com processador *i7-3537* de 2,00Ghz e com 8GB de memória RAM.

Os testes iniciais foram executar as instâncias interessantes informadas pelo monitor Manassés. Em todos os casos, a solução encontrada foi igual a solução desejada. As Figuras de 3 e 4 são algumas dessas instâncias, *in0* e *in2* respectivamente. Para a primeira, demorou-se 15 milisegundos e já para a segunda 19 milisegundos.

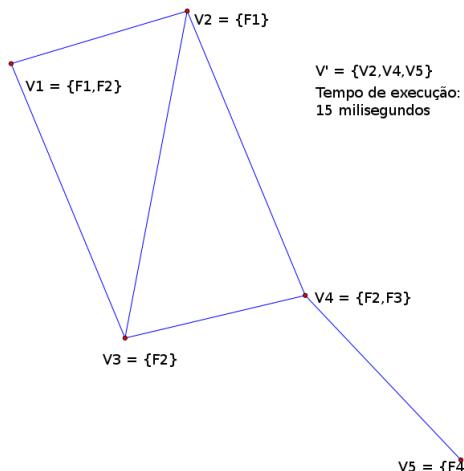


Figure 3. Instância *in0*

A implementação também foi executada para algumas instâncias de testes, como por exemplo: 17 vértices e 45 arestas, 30 vértices e 58 arestas, 50 vértices e 81 arestas, e a de 100 vértices e 194 arestas, todos eles com 5 focos. A Figura 5 é uma dessas instâncias

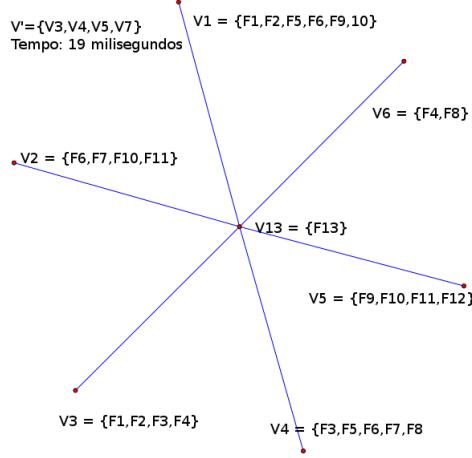


Figure 4. Instância *in2*

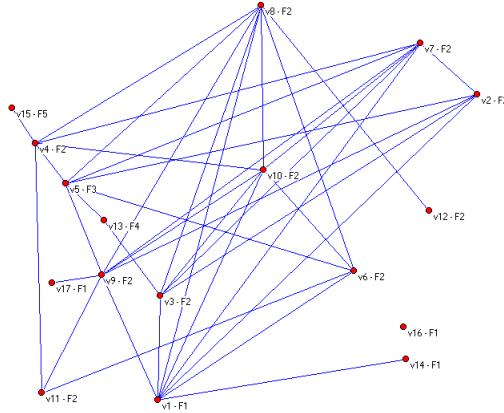


Figure 5. Instância de teste com 17 vértices e 45 aresta e 5 focos

de teste, com 17 vértices e 45 arestas e 5 focos, possuindo uma solução de $\{v_1, v_3, v_4, v_5, v_{13}, v_{15}\}$.

Para observar o tempo de execução do algoritmo, foram feitos mais teste com grafos esparsos variando de 5 vértices a até 300 vértices (por consequência o tamanho de arestas também aumentava). Com base dos dados capturados foi criado a Figura 6 que é um gráfico que nos exibe o tempo de execução de um algoritmo em função, principalmente do aumento da quantidade de vértices. Pode-se observar pelo gráfico que o tempo cresce muito rapidamente para uma quantidade de 200 vértices, o que era de se esperar, pois quanto mais vértices, mais árvores serão necessárias criar, mais cortes terão de serem feitos (principalmente se a quantidade de focos é muito menor que a quantidade de vértices).

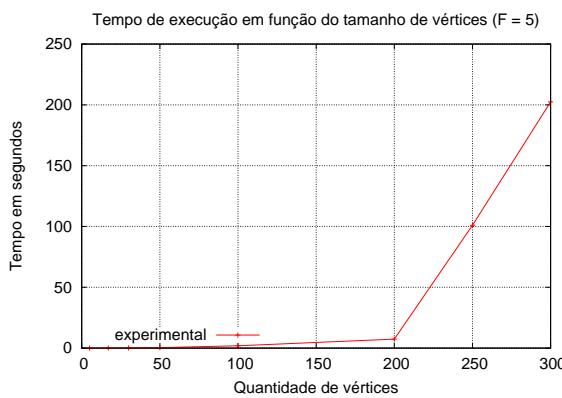


Figure 6. Tempo de execução em função do aumento de vértices em um grafo esparsos

A maior execução feita foi com 300 vértices e 24235 arestas e com 5 focos, levando um tempo de aproximadamente 4 minutos.

6. Exercício 6 - Compare a análise e a execução, respectivamente, Exercícios 3 e 5. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

Para comparar se as previsões teóricas estão de acordo com os resultados obtidos experimentalmente, foi criado a Figura 7 que representa um gráfico com o tempo teórico estimado em comparação com o resultado experimental. Para isso, foi considerado que o computador utilizado de 2GHz realiza 2×10^9 operações por segundo.

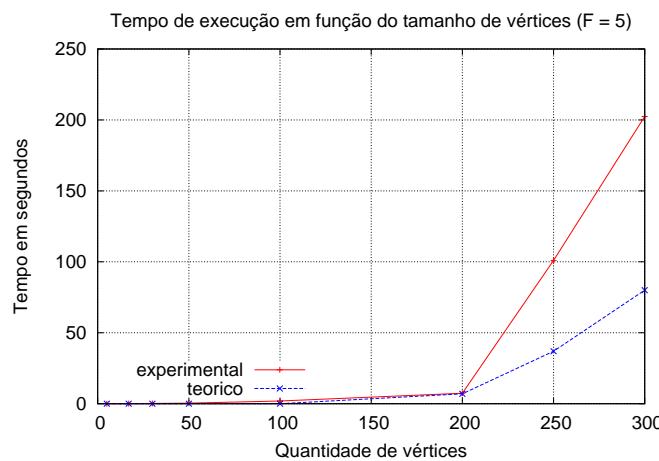


Figure 7. Comparação entre o tempo de execução teórico e experimental

Apesar dos valores da previsão teórica estarem abaixo do resultado experimental, essa diferença já era esperada, uma vez que alguns detalhes de como por exemplo o custo de operação de estruturas como o de *ArrayList* foram desconsideradas. E a diferença observada também deve ser atribuída ao tipo de grafo testado, grafos conexos fará uma quantidade de operações diferentes quando comparado com grafos

desconexos por exemplo. Como estamos tratando de uma complexidade assintótica, a expressão $O(V^3F + V^2EF)$ significa que existirá uma constante c tal que a propriedade $custo_real \leq c*(V^3F + V^2EF)$ seja real, e ao que indica, é possível achar uma constante c para satisfazer essa igualdade.

Como podemos ver, o comportamento das duas curvas são parecidas e mantêm um formato semelhante. Por exemplo, até uma quantidade de 200 vértices as duas curvas eram praticamente iguais, e para depois de 200 vértices, ambas cresceram bastante embora que tenha sido para taxas um pouco diferentes. O que indica que a expressão encontrada é bem próxima da expressão real.

Referências

Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

Newman, M. E. J. (2003). The structure and function of complex networks. *SIAM REVIEW*, 45:167–256.

A. Código Fonte

A seguir serão mostradas três classes implementadas em Java para o problema. A primeira é a classe responsável pela definição de grafo, a segunda responsável pela definição de uma lista de adjacência, e por fim, a classe responsável por encontrar os vértices ótimos para a solução do problema proposto.

A.1. Classe TAD_Grafo - Estrutura do Grafo

```
import java.util.ArrayList;

public class TAD_Grafo {

    private ArrayList<ListaAdjacencia> adj;
    private int n_Vertices;
    private int qnt_focos;

    public TAD_Grafo (int n_Vertices) {
        this.n_Vertices = n_Vertices;
        adj = new ArrayList<ListaAdjacencia>(n_Vertices);
        ListaAdjacencia temp;
        for(int i=0;i<n_Vertices;i++){
            temp = new ListaAdjacencia(i);
            adj.add(temp);
        }
        qnt_focos =0;
    }

    public int n_Vertices() {
        return this.n_Vertices;
```

```

}

public int contemVertice(int v) {
    if(adj.contains(v)) return v;
    return -1;
}

public ArrayList<Integer> getAdjs(int v) {
    return adj.get(v).getAdjacencias();
}

public void insereAdj(int indice,int v) {
    adj.get(indice).addAdjacencias(v);
}

public void insereAresta_U(int v1, int v2) {
    adj.get(v1).addAdjacencias(v2);
    adj.get(v2).addAdjacencias(v1);
}

/*
 * Metodos referentes ao focos de cada vértice
 */

public void setQntFocos(int f) {
    qnt_focos=f;
}
public int getQntFocos() {
    return qnt_focos;
}
public void insereFoco(int vertice, int foco) {
    adj.get(vertice).insereFoco(foco);
}

public ArrayList<Integer> getFocos(int vertice) {
    if(vertice<=n_Vertices)
        return adj.get(vertice).getFocos();
    return null;
}
}

```

A.2. Classe ListaAdjacencia - Estrutura de uma das listas de um vértice

```

import java.util.ArrayList;

public class ListaAdjacencia {
    private int vertice;

```

```

private ArrayList<Integer> adjacencias;
private ArrayList<Integer> focos;

public ListaAdjacencia(int v) {
    //Só criar a Lista de um vértice
    vertice = v;
    adjacencias = new ArrayList<Integer>();
    focos = new ArrayList<Integer>();
}

public ListaAdjacencia(int v1, int v2) {
    //Criar a Lista de um vertice e add um vizinho.
    vertice = v1;
    focos = new ArrayList<Integer>();
    adjacencias = new ArrayList<Integer>();
    addAdjacencias(v2);
}

public int getVertice(){
    return vertice;
}

public ArrayList<Integer> getAdjacencias() {
    return adjacencias;
}

public void addAdjacencias(int v_adjacencia) {
    if(!adjacencias.contains(v_adjacencia))
        adjacencias.add(v_adjacencia);
}

public boolean contem_adj(int adjancencia){
    if(!adjacencias.isEmpty())
        if (adjacencias.contains(adjancencia))
            return true;
        return false;
}

/*
 * Metodos referentes à gerênciia dos focos de um vertice
 */

public int getQntFocos() {
    return focos.size();
}

```

```

public void insereFoco(int foco) {
    focos.add(foco);
}

public ArrayList<Integer> getFocos () {
    if(focos.isEmpty()) return null;
    else return focos;
}
}

```

A.3. Classe CoberturaDeFocos - Responsável por encontrar a solução do problema

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.Queue;

public class CoberturaDeFocos {
    private TAD_Grafo graph;
    private int Focospv[];
    private int FocosQntv[];

    public CoberturaDeFocos (TAD_Grafo graph) {
        this.graph = graph;
    }

    public ArrayList<Integer> find_group(int s) {
        /*
         * Declaração e inicialização de variaveis - O(V)
         */

        int v, w;
        Queue<Integer> Q = new LinkedList<Integer>();
        ArrayList <Integer> v_induced = new ArrayList<Integer>();
        int nivel[] = new int[graph.n_Vertices()];
        int antecessor[] = new int[graph.n_Vertices()];
        int conta = 1;
        int max = 0;
        int Focos_Induced[] = new int [graph.getQntFocos()];
        for (v = 0; v < graph.n_Vertices(); v++)     nivel[v] = -1;
        for (v = 0; v < graph.getQntFocos(); v++)
            Focos_Induced[v] = 0;

        /*
         * BFS
         */
        nivel[s] = conta++;
        ArrayList<Integer> focos_w = graph.getFocos(s);

```

```

for(int y=0;y<focos_w.size();y++) {
    Focos_Induced[focos_w.get(y)-1]++;
    if(!v_induced.contains(s+1)) v_induced.add(s+1);
}
boolean ncompleto = true;
boolean aumentou = false;
Q.add(s);
while (!Q.isEmpty() && ncompleto) {
    v = Q.poll();
    aumentou = false;
    ArrayList<Integer> adj_v = graph.getAdjs(v);
    for (int z = 0; z <adj_v.size(); z++) {
        w = adj_v.get(z);
        if (nivel[w] == -1) { // só o computa na 1ª vez
            Q.add(w);
            aumentou = true;
            antecessor[w]=v;
            nivel[w] = conta;
            focos_w = graph.getFocos(w);
            //acha o nivel maximo da arvore criada
            if(max<nivel[w]) max = nivel[w];
            if(focos_w !=null)
                for(int y=0;y<focos_w.size();y++) {
                    Focos_Induced[focos_w.get(y)-1]++;
                    if(!v_induced.contains(w+1))
                        v_induced.add(w+1);
                }
        }
    }
    if(aumentou) conta++;
    ncompleto = false;
    for (v = 0; v < graph.getQntFocos(); v++)
        if(Focos_Induced[v]==0) ncompleto = true;
}

/*
 * Parte que irá cortar as folhas/galhos
 *      desnecessárias da árvore
 */
boolean dispensavel = true;
boolean algo_mudou = true;
while(algo_mudou) {
    algo_mudou = false;
    for(int j=0;j<nivel.length;j++) {
        if(nivel[j]>=1){
            if(nivel[j]==max) {

```

```

dispensavel = true;
ArrayList<Integer> f = graph.getFocos(j);
for(int y=0;y<f.size();y++) {
    if( Focos_Induced[f.get(y)-1] ==1) {
        dispensavel = false;
    }
}
if(dispensavel){
    int quant_filhos =0;
    //verifica se o vértice pai tem outro filho
    for(int x=0;x<antecessor.length;x++)
        if(antecessor[x] ==antecessor[j]){
            // se algum vertice ainda é filho dele,
            // nao pode deletar
            quant_filhos++;
        }

    if(quant_filhos==1){ //filho unico
        nivel[antecessor[j]]=max;
    }
    for(int i =0;i<v_induced.size();i++)
        if(v_induced.get(i)== (j+1))
            v_induced.remove(i);
    for(int y=0;y<f.size();y++)
        Focos_Induced[f.get(y)-1]--;
    nivel[j]=-1;
    antecessor[j] =-1;
    algo_mudou = true;
}
} else if(nivel[j]!=1){
    int quant_filhos =0;
    for(int x=0;x<antecessor.length;x++)
        if(antecessor[x] ==j){
            // se algum vertice ainda é filho dele,
            // nao pode deletar
            if(x!=j){
                quant_filhos++;
            }
        }

    if(quant_filhos==0){
        dispensavel = true;
        ArrayList<Integer> f = graph.getFocos(j);
        if(f != null)
            for(int y=0;y<f.size();y++) {
                if( Focos_Induced[f.get(y)-1] ==1) {

```

```

                dispensavel = false;
            }
        }
        if(dispensavel) {
            for(int i =0;i<v_induced.size();i++)
                if(v_induced.get(i)== (j+1))
                    v_induced.remove(i);
            if(f != null)
                for( int y=0;y<f.size();y++)
                    Focos_Induced[f.get(y)-1]--;
            nivel[j]=-1;
            antecessor[j] =-1;
            algo_mudou = true;
        }
    }
}
}
}
}

ncompleto = false;
for (v = 0; v < graph.getQntFocos(); v++)
    if(Focos_Induced[v]==0) ncompleto = true;
if(ncompleto) return new ArrayList<Integer>();
return v_induced;
}

public ArrayList<Integer> solution_ListAdj() {
    TAD_Grafo graph_sol = new TAD_Grafo(graph.n_Vertices());
    //Mapear a quantidade em ordem descrecente os
    ArrayList <Integer> v_induced = new ArrayList<Integer>();
    v_induced =find_group(0);
    for (int i=1;i<graph.n_Vertices();i++) {
        ArrayList <Integer> v_induced_tmp =find_group(i);
        if(v_induced.isEmpty())
            v_induced = v_induced_tmp;
        else if (v_induced_tmp.isEmpty()){
            v_induced = v_induced;
        }else{
            if(v_induced_tmp.size()<v_induced.size())
                v_induced = v_induced_tmp;
            else if(v_induced_tmp.size()==v_induced.size()) {
                int tmp1 =0;
                int tmp2 =0;
                for(int t=0;t<v_induced.size();t++){
                    tmp1 = tmp1+v_induced_tmp.get(t);
                    tmp2 = tmp2+v_induced.get(t);
                }
                if(tmp1>tmp2)
                    v_induced = v_induced_tmp;
                else
                    v_induced = v_induced;
            }
        }
    }
    return v_induced;
}

```

```
        }
        if (tmp1<tmp2)
            v_induced = v_induced_tmp;
    }
}
Collections.sort(v_induced);
return v_induced;
}
}
```

Projeto e Análise de Algoritmos – Trabalho prático de grafos - ZikZeroZ

Vinícius Silva Barros

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
(UFMG)
Belo Horizonte – MG – Brazil
viniciusbarros@dcc.ufmg.br

Abstract: This study aims to show the definition and possible solution for an instance of the set-cover problem. The goal is to choose the minimum number of vertices of a given graph so that one can access the entire universe mosquito breeding spots and keep the resulting graph connected. Such solution is composed of an algorithm along with its complexity function and several tests are run in order to argue and evidence the exponential behaviour of the optimal solution for any set-cover problem.

Resumo. Este documento apresenta a definição e resolução do problema set-cover, ilustrado na forma de um grafo de colaboradores cujo objetivo é selecionar o número mínimo de vértices de tal forma que seja possível acessar todo um universo de focos do mosquito *Aedes Aegypti* e ao mesmo tempo garantir que o grafo resultante seja conectado. É proposto um algoritmo, juntamente com sua função de complexidade e são realizados testes que comprovam o comportamento exponencial da solução ótima do set-cover.

1. Introdução

Este documento apresenta a resolução do trabalho prático de grafos proposto na disciplina de PAA. Cada seção contém uma questão diferente do exercício. O código foi entregue juntamente com este documento e está comentado em suas partes mais importantes. Além disso, foram entregues também os arquivos de entrada utilizados na realização de testes.

2. Modelagem do problema

O problema apresentado envolve, intrinsecamente, a aplicação de grafos desde sua definição mais fundamental, onde é descrita uma rede baseada em voluntários e laços de amizade. No problema, cada voluntário corresponde a um vértice e cada laço de amizade corresponde a uma aresta, como apresentado na figura 1. Entende-se que o grafo gerado é não direcionado e suas arestas não possuem pesos. O grafo não direcionado implica essencialmente que se o voluntário v possui amizade com o voluntário u , então a recíproca é verdadeira e o voluntário u possui amizade com o voluntário v .

Os parâmetros de entrada do problema são os voluntários v , as relações de amizade entre cada voluntário e o conjunto F de focos do mosquito *Aedes Aegypti*, responsável pela transmissão do vírus Zika. É informado ainda que para cada voluntário v é definida uma relação $R(v): V \rightarrow F$ que indica o conjunto de focos aos quais o usuário possui acesso.

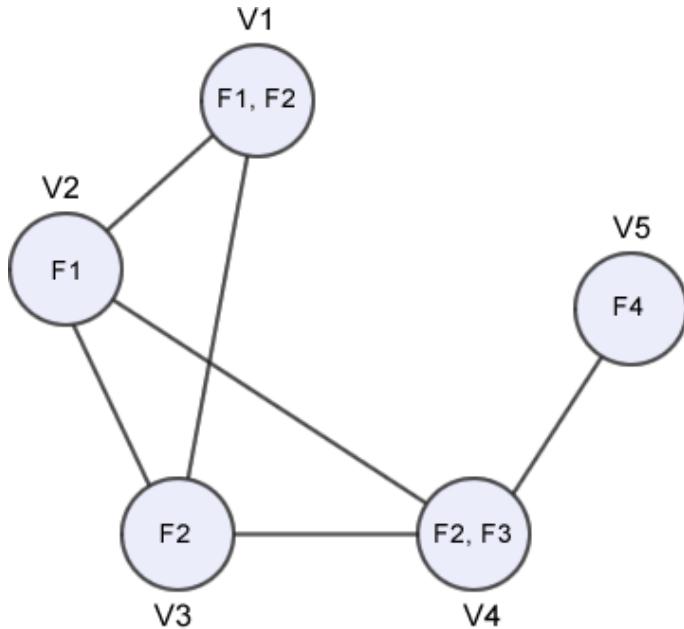


Figura 1. Instância de exemplo do problema ZikaZeroZ. Cada vértice corresponde a um voluntário e as arestas apresentam as relações entre os voluntários. Os números em cada vértice informam os focos aos quais aquele vértice possui acesso.

Neste cenário, o problema consiste em descobrir o menor número de voluntários que permite acessar todos os focos do Zika Vírus. Além disso, os voluntários selecionados precisam formar um grafo conexo, ou seja, devem formar uma única árvore. Os principais parâmetros do problema são:

- V representa o conjunto de vértices onde $|V| = n$.
- A representa o conjunto de arestas onde $|A| = m$.
- F representa o conjunto de focos onde $|F| = r$.
- $R(v): V \rightarrow F$ é a relação entre os vértices e os focos do mosquito, definido para cada vértice $v \in V$ e $R(v) \subseteq F$

2.1. Redução para o problema do set-cover

De acordo com (Cormen, Leiserson, Rivest, & Stein, 2011), uma instância do *set-cover* consiste em um conjunto finito X (também chamado universo) e uma família F de subconjuntos de X de tal forma que todos os elementos de X pertencem a pelo menos um subconjunto em F . O problema do *set-cover* é encontrar um subconjunto mínimo contido em F de forma que a união de todos os seus membros forme X . Este problema é NP-Completo.

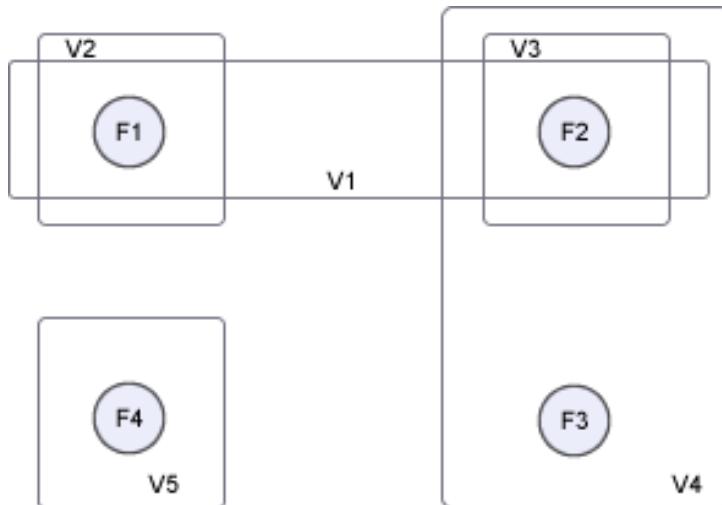


Figure 2. Representação do grafo da Figura 1 como uma instância do set-cover.
Cada vértice representa um conjunto e cada foco do Zika Vírus representa um elemento do universo.

Na Figura 2 ilustramos o problema do *set-cover* com a entrada apresentada para o ZikaZeroZ. Neste caso, omitimos as arestas que identificam as relações entre os conjuntos para fins de visualização. Cada conjunto é definido pela relação R apresentada na definição do problema e a união entre todos os conjuntos forma o universo F dos focos. Os subconjuntos são definidos por $R(v_1) = \{F_1, F_2\}$, $R(v_2) = \{F_1\}$, $R(v_3) = \{F_2\}$, $R(v_4) = \{F_2, F_3\}$, $R(v_5) = \{F_4\}$.

A resposta ótima do *set-cover* consiste em encontrar os subconjuntos que satisfazem a equação 1, onde C representa o subconjunto de menor tamanho possível.

$$X = \bigcup_{F \in C} F \quad \text{Eq. 1}$$

2.2. Modelagem do problema

A modelagem do problema consiste em percorrer, sistematicamente, cada vértice e cada aresta do grafo buscando pelos subconjuntos que satisfazem a equação 1. Entendendo que o problema é NP-completo (Cormen, Leiserson, Rivest, & Stein, 2011) e que a solução apresentada deve ser ótima, a abordagem mais direta para encontrar o conjunto com a resposta é computar todas as possibilidades de conjuntos até encontrar a solução ótima, ou seja, aquela com menor tamanho.

Colocando em termos do problema *set-cover*, a abordagem escolhida para a solução consiste em encontrar todas as combinações dos conjuntos formados por $R(v)$ e verificar quais dessas combinações formam o universo F . Quando alguma combinação formar o universo, percorremos o grafo G para verificar se os vértices selecionados formam um grafo G' conexo.

Dados informados/conhecidos

- i) O grafo G é não direcionado e não possui pesos nas arestas.
- ii) O conjunto de vértices V (voluntários).
- iii) O conjunto de arestas A (relações de amizade).
- iv) O conjunto de focos F .
- v) A relação entre os vértices e os focos, definida por R .

Suposições e/ou limitações

- i) Não existe uma solução ótima conhecida em tempo polinomial pois o problema é NP-Completo (Cormen, Leiserson, Rivest, & Stein, 2011).
- ii) O conjunto F dos focos representa o universo que se deseja cobrir na analogia com o *set-cover*.
- iii) Cada subconjunto de focos obtido pela relação $R(v): R \rightarrow F$ é um subconjunto de F .
- iv) O caso trivial corresponde ao conjunto de todos os vértices V . Este conjunto sempre será capaz de gerar o universo F .
- v) O problema sempre irá apresentar uma solução para as entradas utilizadas, mesmo que a solução seja a trivial.
- vi) Os vértices estão enumerados de 1 até n , as arestas estão enumeradas de 1 até m . Os focos estão numerados de 1 até r . Todas as numerações citadas são sequenciais e com números inteiros. Internamente os dados são armazenados de forma ordenada crescente.
- vii) Cada vértice possui uma propriedade $v.f$ que informa a lista de focos aos quais possui acesso.
- viii) A solução escrita na saída deve estar em ordem crescente de acordo com o índice do vértice.

3. Algoritmo

Sabendo que não existe solução ótima em tempo polinomial, de acordo com a limitação (i), é possível concluir que o algoritmo que resolve o problema precisa percorrer todas as possíveis combinações de n vértices até encontrar uma combinação V' que forma o universo F . Ao encontrar tal combinação, é feita uma busca em largura no grafo $G' = (V', E')$ para verificar sua conectividade e, em caso positivo, o conjunto V' é armazenado como possível candidato a solução, faltando apenas percorrer as demais combinações de forma a garantir que $|V'| = \min$ (a cardinalidade de V' é a menor possível).

De acordo com o dado informado (i), o grafo G é não direcionado, o que implica que a quantidade de caminhos possíveis é dada através da combinação entre os vértices tomadas de k elementos onde $k \in [1..n]$. Para o grafo apresentado na figura 2 é necessário analisar todas as combinações de 5, 4, 3, 2 e 1 vértices tomadas no conjunto total de 5 vértices. Este exemplo possui um total de 31 combinações, que pode ser reescrito na forma $2^5 - 1$.

Para um conjunto de n vértices, a equação 2 define a função $c(n)$ que informa a quantidade total de combinações possíveis. A prova desta equação não faz parte do

escopo deste trabalho mas o resultado pode ser facilmente obtido através do Binômio de Newton (Weisstein). É importante observar que $i \in [1..n]$ dado que a combinação com zero elementos não interessa na resolução deste problema.

$$c(n) = \sum_{i=1}^n \binom{n}{i} = 2^n - 1 \quad \text{Eq. 2}$$

Para verificar se V' forma um grafo conexo é suficiente utilizar uma busca em largura no grafo $G' = (V', E')$ e garantir que todos os vértices V' tenham sido visitados. Sob o aspecto técnico da solução implementada, a distância e a referência para o vértice pai (v, π) retornadas por BFS não são importantes. Por este motivo o algoritmo é reescrito de forma a desconsiderar estes parâmetros e apenas marcar os vértices alcançáveis como visitados. Ao aplicar BFS podemos escolher qualquer vértice no conjunto V' como a fonte.

3.1. Escrevendo as combinações

Certamente, um dos aspectos técnicos da implementação e influenciam diretamente na eficiência do algoritmo é a metodologia utilizada para escrever as combinações dos elementos do conjunto de vértices. É possível encontrar as combinações de forma iterativa e recursiva, sendo esta última menos recomendada considerando o comportamento exponencial do problema, apresentado pela equação 2. Na implementação deste algoritmo as combinações foram geradas com base em uma relação binária de n bits trabalhando com os ponteiros dos vértices.

De acordo com a suposição (vi) os vértices estão armazenados de forma ordenada crescente, sendo esta limitação necessária para o correto funcionamento do algoritmo porque é fundamentalmente mais eficiente começar a verificar os conjuntos de vértices em ordem crescente de combinações. Ou seja, verificamos inicialmente todas as combinações formadas por $1, 2, \dots, n$ vértices nesta ordem. Dessa forma, ao encontrarmos um conjunto V' que forme ambos o universo F e um grafo G' conexo, sabemos que qualquer outra combinação que satisfaça às mesmas condições deverá ter uma cardinalidade maior ou igual a $|V'|$ e, neste ponto, podemos encerrar a execução do algoritmo e retornar o conjunto encontrado, dado que este é o menor possível.

Em termos matemáticos, considere as combinações tomadas em k elementos onde $k \in [1, \dots, n]$ nesta ordem. Seja um conjunto V' tal que $|V'| = k$. Se V' satisfaz a equação 1 e forma um grafo G' conexo para um valor $k = i$, então nenhum outro valor de $k < i$ satisfaz às mesmas condições. Neste caso, a execução termina com $k = i$ e o resultado é retornado. Este é o invariante de *loop* utilizado na prova de que o algoritmo funciona. Como critério de desempate, foi considerado que a primeira combinação encontrada será retornada (pois é a menor, dado que o algoritmo funciona). Para esta combinação, a soma dos índices dos vértices será a menor possível **entre as soluções com mesmo número de elementos** porque os vértices foram considerados em ordem crescente de numeração pela suposição (vi).

3.2. Apresentação do algoritmo

O algoritmo desenvolvido para a solução do problema é razoavelmente simples e pode ser escrito em poucas linhas. É certo que a implementação utilizada tem um impacto na performance, mas é de se esperar que, para uma implementação razoável, a ordem de complexidade não seja substancialmente alterada. A seguir está apresentado o algoritmo SET-COVER-GENERAL (G, F).

```
1   SET-COVER-GENERAL (G, F)
2       for each ordered combination  $V'$  of  $G.V$ 
3           if  $V'$  makes the universe  $F$  and  $V'$  forms a connected  $G'$ 
4               return  $V'$ 
5
6       return NIL;
```

O trabalho mais pesado do algoritmo acontece nas linhas 2 e 3, onde é necessário encontrar as combinações V' de V e verificar se determinado V' atende aos critérios da solução. O algoritmo que calcula as combinações não faz parte do escopo deste trabalho e por isso não será apresentado. O algoritmo que verifica se uma combinação V' forma o universo F é definido como MAKES-UNIVERSE(V', F) e retorna TRUE em caso positivo.

```
1   MAKES-UNIVERSE ( $V', F$ )
2       universe[F.length] = false //Vetor do tamanho do universo (r)
3
4       for v in  $V'$ 
5           for f in v.f
6               universe[f] = true
7
8       for i = 0 .. F.length
9           if universe[i] = false
10          return false;
11
12      return true
```

O algoritmo MAKES-UNIVERSE(V', F) funciona criando um vetor de *flags* que tem o tamanho do universo a ser analisado (linha 2) cujo valor inicial é de FALSE para todas as posições. Para cada novo foco encontrado, o *flag* correspondente àquela posição é definido como TRUE (linha 6). A suposição (vii) garante que cada vértice possui a propriedade $v.f$ (linha 5) e a suposição (vi) garante que esta metodologia funciona porque os focos estão ordenados em forma crescente. Na linha 8 é feita uma verificação passando por todos os elementos de *universe* e, se algum dos elementos for falso, isso significa que V' não foi capaz de cobrir o universo inteiro e então o algoritmo retorna falso. Caso contrário, o valor retornado é verdadeiro.

O algoritmo **BFS-CONNECTED(G' , s)** é uma variação da busca em largura que verifica se o grafo G' é conexo, ignorando o parâmetro de distância e o antecessor. Neste algoritmo é considerado que cada vértice possui uma propriedade `v.visited`.

```

1  BFS-CONNECTED( $G'$ ,  $s$ )
2      Q = empty queue
3
4      for v in  $G'$ 
5          v.visited = false
6
7      Q.enqueue( $s$ )
8
9      while(Q != empty)
10         current = Q.dequeue()
11
12         for u in current.adj
13             if u.visited = false
14                 u.visited = true
15                 Q.enqueue(u)
16
17         for v in  $G'$ 
18             if v.visited = false
19                 return false
20
21     return true

```

Na implementação do **BFS-CONNECTED(G' , s)**, certamente o preço de se criar um novo grafo G' supera aquele de simplesmente adicionar um novo *flag* para cada vértice de G informando se este faz parte ou não do novo grafo. Por este motivo, o que foi apresentado no código é uma variação que recebe como parâmetros o grafo original G , a lista V' e a fonte s . Tal implementação não altera a ordem de complexidade de **BFS-CONNECTED(G' , s)**.

Por fim, o algoritmo final **SET-COVER(G, F)** está apresentado a seguir.

```

1  SET-COVER( $G, F$ )
2
3      for k = 1 .. n
4          for each  $V'$  as a combination of  $G$  in  $k$  elements
5              if MAKES-UNIVERSE( $V', F$ )
6                  source = first vertex of  $V'$ 
7
8                  if BFS-CONNECTED( $G, V', source$ )
9                      return  $V'$ 
10
11     return NIL

```

O algoritmo recebe como parâmetros o grafo original G esperando que cada vértice v contenha a propriedade $v.f$ pela suposição (vii). O loop principal (linha 3) se mantém pelo seguinte invariante:

Não existe nenhum subconjunto V' com $|V'| < k$ que forme o universo e que forme um grafo conectado.

Início: antes de começar a primeira iteração, $k = 1$ e o caso trivial é válido pois não existe nenhuma combinação que satisfaça aos requisitos da solução de tal forma que $|V'| = 0$.

Manutenção: durante a execução dos loops internos, a linha 5 verifica se dada combinação forma o universo F e, em caso positivo, a linha 8 verifica se esta combinação forma um grafo conexo. Se o grafo for conexo, seu valor é retornado e existe uma combinação com k elementos que forma F , o que mantém o invariante válido pois $|V'| = k > k - 1$. Caso contrário, percorremos as demais combinações sem alterar o valor de k no loop interno e, após a verificação exaustiva da linha 4, o valor de k é incrementado e o loop externo executa uma nova iteração.

Terminação: ao fim do algoritmo, $k = n + 1$ e se a linha 11 for alcançada isso implica que não existe nenhuma combinação com qualquer valor de $k \in [1..n]$ que satisfaça aos critérios de solução. Caso contrário a solução teria sido retornada anteriormente na linha 9.

4. Análise de complexidade

Como já foi apresentado anteriormente, a complexidade do algoritmo é no mínimo exponencial pois a suposição (i) determina que o problema é NP-Completo e a definição do trabalho exige uma solução ótima. A equação 2 mostra que o número de combinações que precisam ser analisadas no pior caso é de $\text{comb}(n) = 2^n - 1$, que acontece na linha 4 do **SET-COVER(G, F)**. O que irá influenciar no restante da análise é o processo que acontece nas linhas 5 e 8 do algoritmo. As complexidades dos algoritmos utilizados estão apresentada a seguir.

- A complexidade do algoritmo **MAKES-UNIVERSE(V', F)** é de $O(n \times r)$ pois de acordo com as linhas 4-5, todos os focos de todos os vértices do conjunto V' deverão ser analisados.
- A complexidade do algoritmo **SET-COVER(G, F)** é de $O(n + m)$ pois no pior caso todas as arestas de todos os vértices precisarão ser analisadas.

Portanto, no pior caso, a complexidade total do algoritmo **SET-COVER(G, F)** é de $O(2^n(n \times r + n + m))$ ou, simplificando, vem $O(rn \times 2^n)$. Pode parecer estranho que a quantidade de arestas não influencie na complexidade final do algoritmo, mas uma análise amortizada mostra que, de fato, a quantidade de arestas tem pouco impacto na resolução do *set-cover* pois o problema inicial trata da análise combinatória de conjuntos.

A linha 8 do $\text{SET-COVER}(G, F)$ é executada poucas vezes em relação ao número total de combinações e para cada execução o custo relacionado às arestas é linear. Além disso, o único caso onde o algoritmo $\text{BFS-CONNECTED}(G', s)$ utiliza todas as arestas do grafo original é o caso trivial, onde todos os vértices de G fazem parte de G' . Se quisermos considerar as arestas, uma análise amortizada mostra que a complexidade do algoritmo é $O(rn \times 2^n + n \times m)$.

Quanto à análise de complexidade espacial, o algoritmo precisa essencialmente armazenar todos os vértices, todas as arestas, todos os focos de um grafo G e um grafo G' que, no pior caso, pode ser igual a G . Como na implementação das permutações estamos essencialmente alterando a ordem de ponteiros, não é feito o uso de outros recursos de memória. Dessa forma, armazenando os vértices em uma lista de adjacência e os focos em uma lista como propriedades dos vértices, o gasto de memória do programa é da ordem de $O(2 \times (n + m + r))$, o que equivale a $O(n + m + r)$.

5. Implementação e testes

A implementação do algoritmo foi feita em C++ e o código fonte está disponível juntamente com este documento. Para realização de testes, as entradas iniciais foram dadas juntamente com o problema e, em seguida, foram propostas novas entradas interessantes para verificar a eficiência do algoritmo. Todas as entradas de teste utilizadas foram enviadas juntamente com este documento.

5.1. Aumentando o número de vértices

O primeiro teste consiste em analisar o pior caso do algoritmo, onde todos os vértices precisam estar juntos para formar o universo. Para isso, foram criados grafos onde o número de focos é igual ao número de vértices e cada vértice i possui acesso somente ao foco i , de acordo com a figura 3.

Neste cenário, foram feitos testes com o número de vértices no conjunto $\{4, 6, 9, 16, 18, 20, 22, 25\}$. O resultado está sumarizado no gráfico apresentado na figura 4, onde o tempo de execução foi calculado em milissegundos.

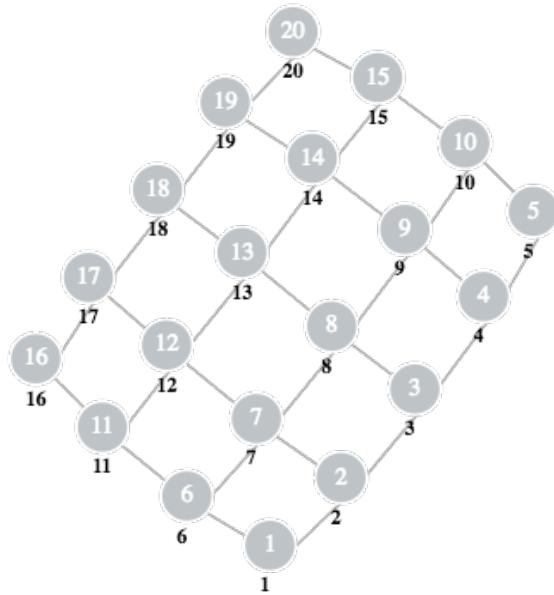


Figura 3. Grafo com 20 vértices e 20 focos onde cada vértice i acessa somente o foco i, representando o pior caso do algoritmo.

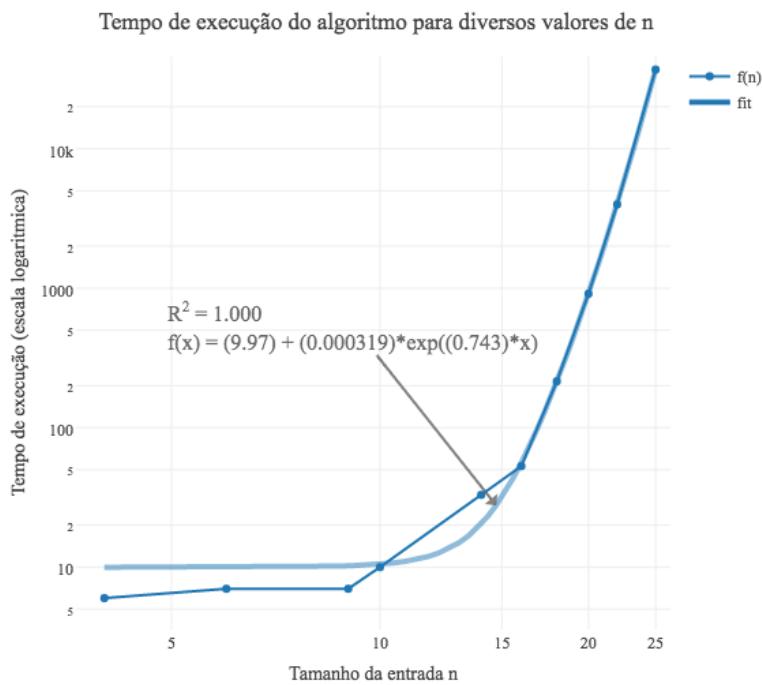


Figura 4. Tempo de execução em ms pelo tamanho da entrada n. O gráfico possui comportamento exponencial na variável n. A quantidade de vértices v é igual à quantidade de focos. A linha mais clara representa curva obtida através de um *fitting-exponential* nos dados.

5.2 Aumentando o número de focos

O próximo teste consiste em manter constante o número de vértices e aumentar o número de focos. Foram utilizados 20 vértices e a variação dos focos está no conjunto $\{20, 40, 60, 80, 100, 120, 140\}$. Novamente, o grafo foi construído para gerar o pior caso no algoritmo.

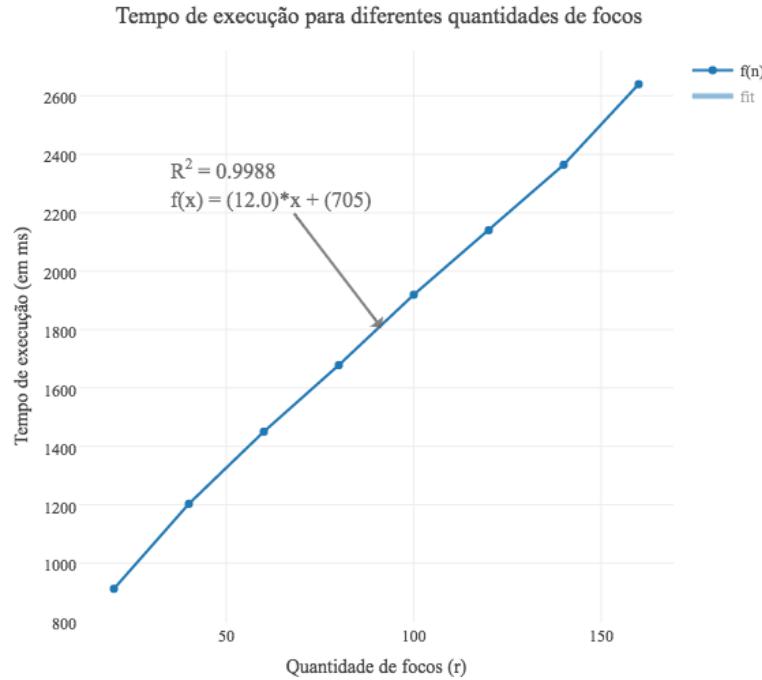


Figura 5. Tempo de execução em milissegundos pelo tamanho da entrada r. O comportamento do algoritmo para variações de r é linear, como indicado pela função de complexidade. A linha mais clara mostra a reta obtida através de uma regressão linear dos dados.

5.3 Distribuição aleatória de focos

O propósito deste teste é verificar como o algoritmo executa dada uma distribuição aleatória dos focos. Foram feitos 6 testes com 24 vértices e 20 focos onde cada vértice pode ter até $t = f/(2 \times \lg f)$. Os grafos gerados, juntamente com as distribuições dos focos estão incluso nos documentos enviados juntamente com este relatório. A figura 6 apresenta o resultado da execução para cada uma das seis entradas juntamente com o grafo resultante.

Durante este teste, o tempo médio para a execução do algoritmo foi de 4202ms e o desvio padrão foi de 2209ms.

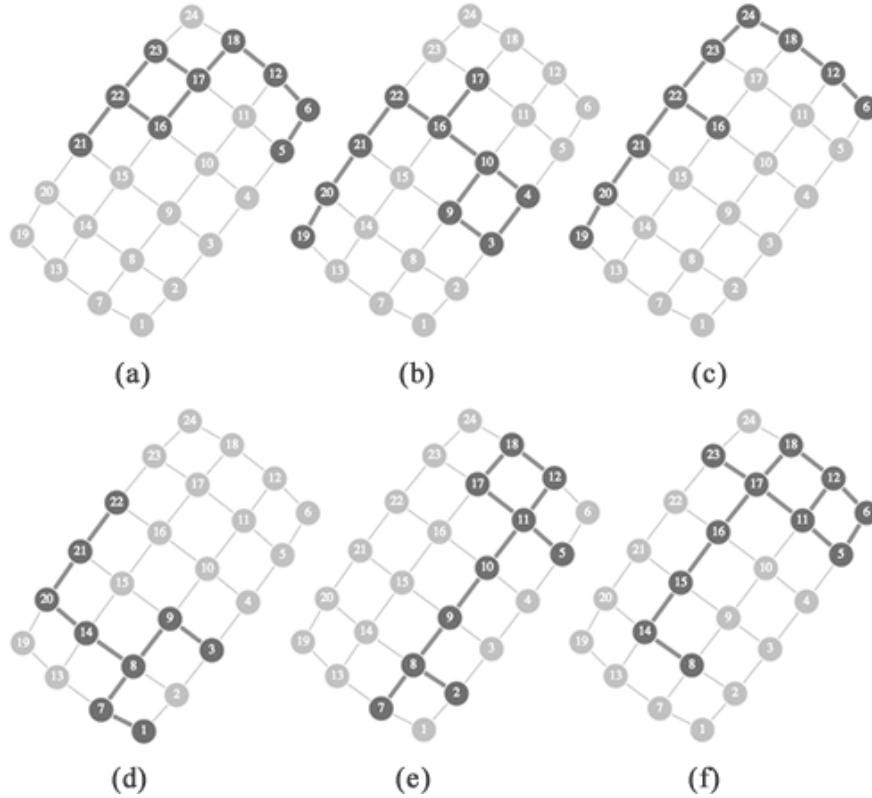


Figura 6. Resultado da execução para entradas aleatórias. A média do tempo de execução para cada entrada é (a) 2784ms; (b) 4217ms; (c) 4714ms; (d) 1495ms; (e) 3955ms; (f) 8041ms.

6. Comparações

A complexidade final do algoritmo foi calculada como sendo $O(2^n(n \times r + n + m))$ com a ressalva de que a análise amortizada permite considerar que o número de arestas apresenta um efeito razoavelmente pequeno no resultado final, podendo ser desconsiderado em alguns casos. Para fins de análise, este trabalho analisou a execução para os piores casos com diferentes quantidades de vértices e focos de mosquito. As conclusões estão sumarizadas a seguir.

A complexidade está limita por um fator exponencial que depende do número de vértices. A figura 4 mostra o tempo de execução para um conjunto de valores de n e um *fitting-exponential* da mesma curva. Este resultado está de acordo com a função de complexidade calculada.

Quanto ao número de focos, a complexidade possui um comportamento linear para variações de r , como mostra a figura 5. Neste caso, o comportamento é tão bem definido que a reta da regressão linear se confunde com os dados adquiridos. Essencialmente, este resultado também está de acordo com a função de complexidade calculada.

É sabido que o algoritmo irá produzir uma resposta qualquer que seja o número de vértices na entrada, mas o problema é executar o procedimento em tempo hábil. A maior entrada para a qual o algoritmo produziu uma saída nessas condições foi com 25

vértices no pior caso, onde a execução gastou 33 segundos. Após este valor, dado o comportamento exponencial, adicionar um outro vértice começaria a gerar saídas absurdamente mais demoradas.

7. Observações importantes

Esta seção contém observações que são válidas no contexto do problema mas que, por não estarem no escopo do trabalho, não serão provadas formalmente ou analisadas de forma detalhada.

- 1) Em um grafo onde todos os vértices estão conectados entre si, a primeira combinação que forma o universo obrigatoriamente irá formar um grafo conexo. Neste caso o número de arestas é da ordem de $|E| = |V|^2$ e, mesmo tendo mais arestas, o tempo de execução do algoritmo pode diminuir substancialmente.
- 2) Existe uma relação ainda não profundamente analisada entre o número de arestas no grafo e a complexidade do algoritmo. Embora seja verdade que mais arestas fazem com que a rotina `BFS-CONNECTED(G', s)` execute mais lentamente, existe em contrapartida uma probabilidade maior de que a mesma retorne `TRUE`, eliminando a necessidade de se realizar futuras buscas pois na primeira vez que `BFS-CONNECTED(G', s)` retorna `TRUE` o algoritmo encontrou uma solução válida.
- 3) Em termos de combinações, melhor caso do algoritmo é aquele onde qualquer vértice individual possui acesso a todos os focos do mosquito, qualquer que seja o vértice. Em termos absolutos, o melhor caso é aquele onde o primeiro vértice tem acesso a todos os focos.
- 4) A implementação das combinações somada ao fato de que a suposição (vi) exige que os vértices estejam armazenados em ordem crescente automaticamente implica na suposição (viii). Por isso não é necessário ordenar os vértices ao encontrar a solução uma vez que os mesmos já se encontram ordenados.
- 5) A única forma de testar a complexidade com base na quantidade de arestas é executar o algoritmo nos casos que as suposições (iv) e (v) em conjunto impedem que existam: muitas combinações devem gerar o universo mas não devem estar conectadas. Em um grafo não direcionado com $|V| - 1$ arestas e sem arestas duplas, o número máximo de vezes que isso pode acontecer está limitado por uma função decrescente $\alpha(|V|)$. Ou seja, quando o grafo possuir $|V|^2$ arestas, obrigatoriamente todas as combinações que formam o universo são também um grafo conexo, como já introduzido pelo item (1) desta seção.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2011). *Introduction to Algorithms*. Cambridge, Massachusetts: MIT Press and McGraw-Hill.

Weisstein, E. W. (s.d.). *Binomial Theorem*. Acesso em 30 de 04 de 2016, disponível em Wolfram MathWorld: <http://mathworld.wolfram.com/BinomialTheorem.html>

Projeto e Análise de Algoritmos

Trabalho Prático 1

ZicaZeroZ

Aluno

Nome

Matrícula

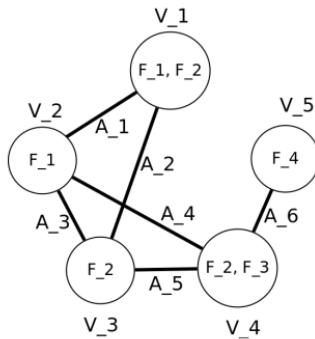
Waner de Oliveira Miranda Miranda 2016672344



Departamento de Ciência da Computação
UNIVERSIDADE FEDERAL DE MINAS GERAIS
Av. Antônio Carlos, 6627 – ICEX Building

1 Questão 1

O problema consiste em encontrar o menor caminho possível do grafo que contemple todos os focos. O grafo utilizado $G(V, E)$ é não-direcional e sem pesos nas arestas. Onde cada vértice possui uma função $R : v \rightarrow [f_1..f_n]$, que indica a relação de focos que cobre, podendo ser um ou mais focos por vértice. Seguindo a mesma representação fornecida durante a especificação do trabalho:



Como um dos requisitos do trabalho, é encontrar a solução ótima do problema, partimos de uma aproximação utilizando força bruta e à procura de pontos para otimização, combinando cada um dos caminhos possíveis criados pelos vértices do grafo. Porém, restringindo as combinações pelos relacionamentos de cada vértice e os focos que este cobre. Ao final verificando a conectividade destes caminhos.

2 Questão 2

Consideramos, com base nas entradas fornecidas para o problema, que o número de focos a serem cobertos é consideravelmente menor do que o número de vértices ou arestas. Logo, os vértices foram agrupados por focos para gerar uma matriz com cada relacionamento. Fazendo a combinação de vértices, segundo a matriz de relacionamento, que cubram todos os focos para formar caminhos candidatos. Após a combinação e criação, os caminhos candidatos são submetidos a outro filtro para selecionar somente os caminhos conexos. Estes caminhos são ordenados pelo menor número de vértices e a menor soma de seus índices.

Foram feitas otimizações no algoritmo para reduzir o tempo de execução e consumo de memória:

- Utilização de filas para evitar o uso de recursão e reduzir o consumo de memória.

- Implementação do armazenamento dos caminhos candidatos utilizando o tipo ”set” do python, reduzindo o número de valores a serem verificados pelo algoritmo que confere a conectividade destes caminhos.

3 Questão 3

Chegamos à um algoritmo com complexidade ciclomática de $O(b^r)$, para r como o número de focos e b a média de vértices que cubram cada foco. A complexidade ciclomática da checagem dos caminhos candidatos é $O(c*(m+n))$, com c sendo o número de caminhos candidatos após o filtro caminhos idênticos e seu valor é próximo a $b!$ ou a fatorial número médio de vértices por foco. Por análise assintótica obtemos a complexidade de $O(b!)$ para esta verificação.

A complexidade espacial é $O(b^r)$, pois este é o numero máximo de caminhos armazenados em memória durante a execução do código.

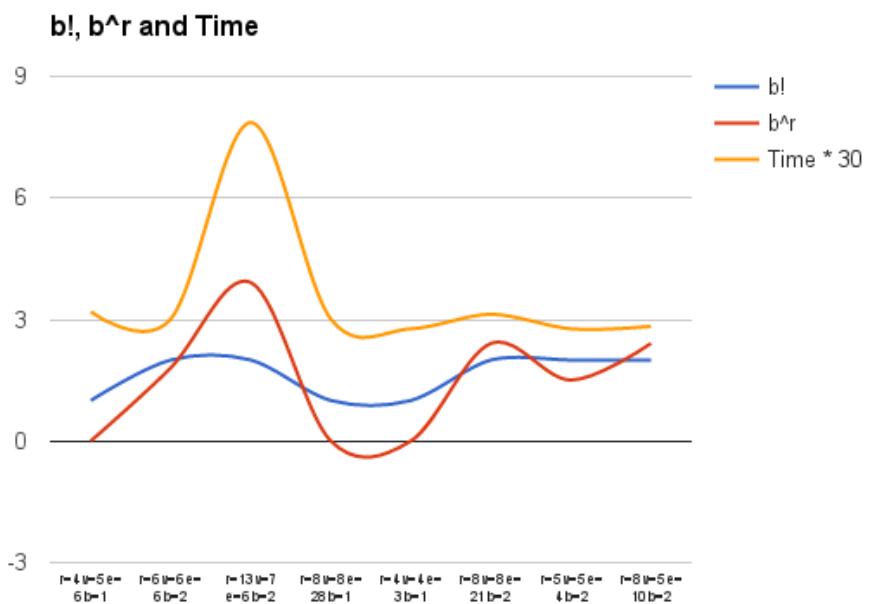
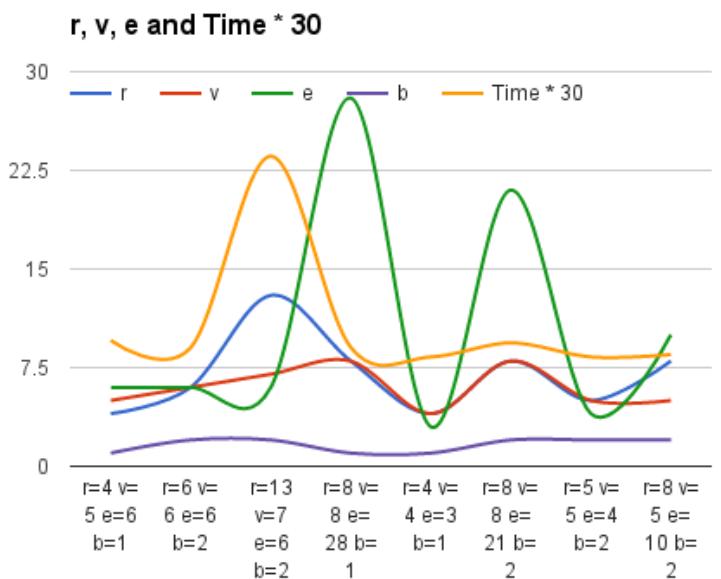
4 Questão 4

O algoritmo está implementado em python 2.7 com 4 arquivos básicos:

- main.py - Para o tratamento de entradas e função principal.
- zicaz.py - Arquivo que contém a classe para solução do problema.
- utilities.py - Somente funções básicas para impressão ou tratamento de tipos de dados.
- graph_utils.py - Contém funções úteis para o tratamento de grafos.

5 Questão 5

Para verificar a relação entre a complexidade ciclomática do algoritmo e os valores pelos quais ela é dada, formulamos um gráfico que exibe para cada execução a relação do tempo e grandeza dos parâmetros e , r e v . Além disso, tomamos a liberdade de multiplicar o valor do tempo, dado em milissegundos, por 30 visando facilitar a visualização comparativa deste no gráfico.



6 Questão 6

Como podemos observar no primeiro gráfico da questão anterior, o maior impacto no tempo, a curva em verde, é dado pelo aumento no valor de r , curva azul. Para lembrar, as complexidades de tempo das partes do algoritmo, estas são respectivamente $O(b^r)$ e $b!$ para criação e avaliação dos caminhos candidatos, onde b é o número médio de vértices por foco. O valor r tem relação direta com o número médio de vértices por foco e também com o número de caminhos possíveis a serem gerados. E o gráfico permite a visualização real do impacto deste nos tempos de execução, conforme havíamos previsto na análise de complexidade. Como a nossa aproximação é baseada no número médio de vértices por foco. Podemos observar dois pontos em que $r = 8$ porém, com tempos desiguais. Para testes o número médio de vértices por foco possui densidade diferente. O segundo gráfico, comparando as funções de complexidade e o tempo obtido, mostra melhor a correlação entre os parâmetros. Pois os picos de tempo entre as funções e o tempo coincidem.

Lista de Exercícios 2 - ZikaZeroZ

Nome: Bruna Vieira Frade

- 1) Exercício 1 [1 ponto]. Apresente uma modelagem para o Problema ZikaZeroZ empregando grafos. Deixe claro quais as restrições e/ou suposições devem ser feitas.

R: O problema representa uma quantidade de grupos de amizade V , no qual cada grupo é um vértice v , que tem acesso a um determinado número de focos f do zika. Cada grupo de amigos v pode ter relações com outros grupos de amigos v_i , essas relações são determinadas através de arestas.

Deve ser encontrado o menor número de relações entre os grupos de amizade que consiga eliminar todos os focos. Lembrando que é a menor quantidade de relacionamentos, ou seja, os vértices do conjunto da solução devem possuir algum relacionamento (arestas que conectam um vértice v_1 a v_2).

Uma estrutura Grafos contendo uma variável v para armazenar o valor do vértice, ou seja, qual o grupo de amigos. Duas listas associadas a cada valor $v \in V$, sendo V o número total de vértices. A primeira lista indexada a v contém todas as suas adjacências, que são as relações de um grupo de amigos com outros grupos. A segunda lista com todos os focos alcançados por v . Ao final a solução apresentada deve ser um subgrafo conexo com o menor número de grupos de amigos para eliminar todos os focos. Dessa forma temos como restrições um resultado para eliminar todos os focos e o menor numero de relações com todos os vértices relacionados

- 2) Exercício 2 [5 3 pontos]. Descreva um algoritmo para resolver o Problema ZikaZeroZ, tendo em vista a modelagem realizada no Exercício 1.

R: Uma possível solução seria um algoritmo que tenta encontrar o menor subconjunto de vértices por tentativa, ou seja, para cada vértice analisa um caminho possível com base em sua adjacência, verificando qual adjacência ainda não foi percorrida e se possui algum foco a ser eliminado, guardando apenas o menor caminho dentre as tentativas.

- 3) Exercício 3 [1 ponto]. Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) do algoritmo proposto no Exercício 2.

R: No Pior caso, a complexidade de tempo seria dada pela função:

$$V * (V * (QdFocosV + QtdAdjV * QtdFocosAdj))$$

Onde V é o total de vértices, $TotalFocos$ é o número total de focos, $QdFocosV$ é a quantidade de focos do vértice V , $QtdAdjV$ é a quantidade de adjacências de V e $QtdFocosAdj$ é a quantidade de focus de cada adjacência de V .

Portanto, temos $O(V^3)$ como complexidade de tempo

No Melhor caso, quando no primeiro vértice a ser pesquisado é encontrado todos os focos, temos a seguinte função de complexidade:

$$V * (QdFocosV + QtdAdjV * QtdFocosAdj)$$

Que seria $\Omega(V)$ Se V possuísse apenas um foco e que por sua vez estivesse no primeiro vértice pesquisado e V não possuísse adjacências.

- 5) Exercício 5 [1 ponto]. Execute testes da implementação do Exercício 4, propondo instâncias interessantes para o Problema ZikaZeroZ. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n , m e r). Outra sugestão é relatar o tamanho da maior instância para o qual foi produzida uma solução.

R:

Teste	Vértice	Aresta	Focos	Tempo (s)
0	5	6	4	-
1	6	6	6	2.17
2	7	6	2	-
3	4	3	2	2.14
4	8	2	2	-
5	5	4	5	2.16
6	5	10	2	-

Foram realizados os testes propostos pelo trabalho, no entanto para determinadas instâncias não foi possível obter o melhor resultado. Mas levando em consideração os grafos nos quais foram obtidos resultados, pode ser observado que quanto maior a quantidade de vértices, focos e arestas, maior é o tempo de execução, provando assim que a complexidade de tempo do algoritmo é variante de acordo com essas entradas.

Projeto e Análise de Algoritmos: Zika Zero Z

Data: 01 de maio de 2016

Roberta Coeli Neves Moreira

Exercício 1

Apresente uma modelagem para o Problema ZikaZeroZ empregando grafos. Deixe claro quais as restrições e/ou suposições devem ser feitas.

O grafo foi modelado tendo os voluntários como vértices, seus laços de amizade como arestas e seus focos de acesso como atributos de cada vértice. Em suma, o grafo G do Problema ZIKAZEROZ foi modelado da seguinte forma:

- Os voluntários constituem os vértices do grafo: \mathbb{V} .
- Os laços de amizade representam as arestas entre os vértices, \mathbb{A} . As arestas serão representadas por uma matriz de adjacência.
- Os focos são representados por números binários, como atributos de cada vértice. O número relativo ao foco indica, em cada dígito na posição i , se o foco i está presente ($i = 1$) ou ausente ($i = 0$) na combinação. A relação entre foco e vértice é representada por $\mathcal{R}(v) = nf$, onde nf é o número que representa a combinação de focos.

A Figura 1 mostra um exemplo de grafo estruturado conforme modelagem descrita acima. Neste grafo, temos 7 voluntários: $\mathbb{V} = \{V1, V2, V3, V4, V5, V6, V7\}$. Os voluntários estão conectados de modo que formam os laços de amizade definidos por $\mathbb{A} = \{A1, A2, A3, A4, A5, A6, A7, A8, A9\}$. Os focos são atributos dos vértices e estão indicados ao lado de cada um deles. Para o vértice $V6$, por exemplo, $F = 011$ indica que os focos 2 e 3 são acessados por $V6$.

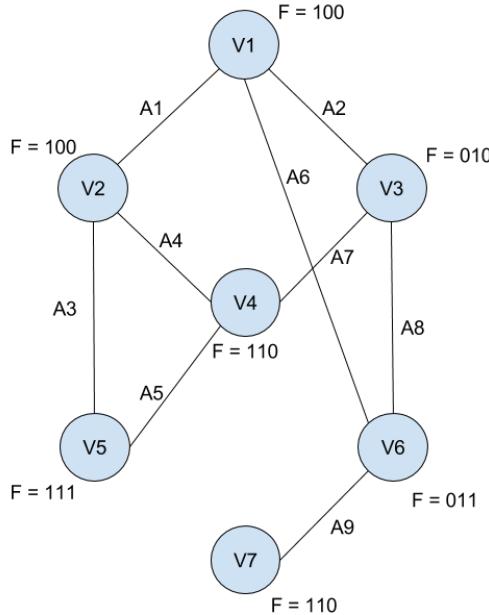


Figura 1: Grafo de exemplo para visualização da modelagem realizada

Objetivo: o problema tem como objetivo em encontrar a combinação $\mathbb{V}' \subseteq \mathbb{V}$ com menor número de vértices para os quais $\mathcal{R}(\mathbb{V}') = \mathbb{F}$.

Suposições e Restrições:

- Cada voluntário $v \in \mathbb{V}$ acessa pelo menos um foco.
- Existe $\mathbb{V}' \subseteq \mathbb{V}$ que tem acesso, conjuntamente, a todos os focos.
- Arestas/laços de amizade são unitárias (não apresentam peso associado).

Exercício 2

Descreva um algoritmo para resolver o Problema ZikaZeroZ, tendo em vista a modelagem realizada no Exercício 1.

O algoritmo para encontrar uma solução ótima para o problema ZIKAZEROZ apresenta os seguinte funcionamento básico:

1. Encontre todas as possíveis combinações de vértices do grafo.
2. Para cada combinação, verifique se o grafo induzido pelos vértices da combinação é conexo.
3. Se o grafo induzido for conexo, verifique se seus vértices contêm todos os focos existentes.
4. Se o grafo induzido for conexo e contiver todos os focos, verificar se a quantidade de vértices é menor que a melhor combinação até o momento.
 - (a) Se a quantidade de vértices for menor que àquela da melhor combinação, a melhor combinação passa a ser aquela dos vértices encontrados na iteração.
 - (b) Se for igual (*critério de empate*), verificar: se a soma dos vértices for menor do que a soma dos vértices da melhor combinação, a melhor combinação passa a ser aquela dos vértices encontrados na iteração.
 - (c) Caso contrário, a melhor combinação permanece inalterada.
5. Repita os passos (2) a (4) para cada combinação.
6. Após iterar sobre todas as possíveis combinações, retorne a melhor combinação (menor número de vértices de menor soma).

O pseudo-código para o problema ZIKAZEROZ é apresentado no Algoritmo 1. Nele, as linhas (2) a (6) fazem a inicialização de algumas variáveis, onde:

- n e r representam o número de vértices e focos, respectivamente.
- $bestNumVertices$ representa a quantidade de vértices da melhor combinação. Esta variável é inicializada com um número elevado (∞), a fim de se encontrar as demais combinações com menor número.
- $bestVertices$ representa a melhor combinação de vértices do grafo. Ela é inicializada com todos os vértices existentes, uma vez que eles representam o pior caso (menor número de voluntários possível = número de vértices do grafo).
- $combinations$ representa o número de combinações possíveis. Para gerar todas as possíveis combinações, podemos considerar que, para cada $v \in \mathbb{V}$, v pode ou não fazer parte da combinação (2 possibilidades). Assim, temos 2^n combinações possíveis, das quais a combinação em que nenhum vértice está presente é eliminada: logo, $2^n - 1$ combinações.

Para cada uma das combinações geradas, ZIKAZEROZ verifica se existe um caminho (grafo induzido) com aquela combinação. Para isso, foi utilizada uma variação da Busca em Largura (*Breadth-First Search (BFS)*), implementada no procedimento *ExistsPath*, apresentado no Algoritmo 3. Se o caminho existir, verifica-se se todos os vértices acessam todos os focos por meio da chamada ao procedimento *HasAllFocuses*. Caso *HasAllFocuses* retorne verdadeiro, compara-se a atual combinação à melhor combinação até o momento (*bestVertices*). Caso a atual combinação seja melhor, *bestVertices* recebe o valor da atual combinação e *bestNumVertices* recebe o número de vértices existentes na mesma. Se for igual (*criterio de empate*), verifica-se: se a soma dos vértices for menor do que a soma dos vértices da melhor combinação, a melhor combinação passa a ser aquela dos vértices encontrados na iteração.

Algoritmo 1: ZikaZeroZ

Data: $G = (\mathbb{V}, \mathbb{A}, \mathbb{F}, \mathcal{R})$

Result: $bestVertices$, where $bestVertices$ is a minimal subset of \mathbb{V} that contains all the focuses in \mathbb{F}

```

1 begin
2    $n \leftarrow \mathbb{V}.length$ 
3    $r \leftarrow \mathbb{F}.length$ 
4    $bestNumVertices \leftarrow \infty$ 
5    $bestVertices \leftarrow$  list containing all vertices
6    $combinations \leftarrow 2^n - 1$ 
7   for  $i \in 1 \rightarrow combinations$  do
8      $combination \leftarrow GenerateCombination(i, n)$ 
9     if ExistsPath( $combination, \mathbb{A}$ ) then
10       if HasAllFocuses( $combination, \mathbb{F}$ ) then
11         if  $combination.numVertices < bestNumVertices$  then
12            $bestNumVertices \leftarrow combination.numVertices$ 
13            $bestVertices \leftarrow combination$ 
14         else if  $combination.numVertices == bestNumVertices$  and
15              $\sum_v combination \leq \sum_v bestVertices$  then
16                $bestVertices \leftarrow combination$ 
17             end
18         end
19       end
20     end
21   end
22 end

```

Os procedimentos invocados por ZIKAZEROZ são listados nos Algoritmos 2, 3 e 4 a seguir. O procedimento **GenerateCombination** (Algoritmo 2) recebe um número inteiro e retorna um *array* de tamanho $|V|$ que é uma representação binária do número de entrada i . Assim, podemos gerar combinações que vão de $i = 1$ até $i = combinations$, alternando-se as possibilidades do vértice em cada posição do *array* existir ou não na combinação.

Algoritmo 2: GenerateCombination

Data: (i, n) , where i is the combination number and n is the number of vertices in the combination

Result: $combination$, the resulting combination

```

1 begin
2   | return array of  $n$  elements which are the binary representation for number  $i$ 
3 end

```

O procedimento **ExistsPath** (Algoritmo 3) permite verificar se o grafo induzido é conexo. Ele implementa uma busca em largura modificada: ele coloca todos os vértices que não estão na combinação em preto (já percorridos) e percorre toda a combinação a partir de um de seus vértices, procurando um caminho que passa por todos os vértices da combinação. Se houver um caminho, todos os vértices estarão em preto e o algoritmo retorna *True*. Caso exista algum vértice que não tenha cor preta, retorna-se *False*.

Algoritmo 3: ExistsPath

Data: (*combination*, \mathbb{V} , \mathbb{A}), where i is the combination of vertices, \mathbb{V} is the array containing all existent vertices, and \mathbb{A} is the adjacency matrix

Result: a boolean that represents if there is a path containing the combination of vertices in *combination*

```

1 begin
2   foreach  $v \in \mathbb{V}$  do
3     if  $v \in \text{combination}$  then
4       |  $v.\text{color} \leftarrow \text{WHITE}$ 
5     end
6     else
7       |  $v.\text{color} \leftarrow \text{BLACK}$ 
8     end
9   end
10   $Q \leftarrow \emptyset$ 
11   $\text{first} \leftarrow \text{take first vertex from } \text{combination}$ 
12   $\text{first.color} \leftarrow \text{GRAY}$ 
13  Enqueue( $Q$ ,  $\text{first}$ )
14  while  $Q \neq \emptyset$  do
15     $u \leftarrow \text{Dequeue}(Q)$ 
16    foreach  $v \in \mathbb{A}[u][v]$  do
17      if  $v.\text{color} == \text{WHITE}$  then
18        |  $v.\text{color} \leftarrow \text{GRAY}$ 
19        | Enqueue( $Q$ ,  $v$ )
20      end
21    end
22     $u.\text{color} \leftarrow \text{BLACK}$ 
23  end
24  foreach  $v \in \mathbb{V}$  do
25    if  $v.\text{color} \neq \text{BLACK}$  then
26      return True
27    end
28  end
29  return False
30 end

```

O procedimento **HasAllFocuses** (Algoritmo 4) percorre cada vértice da combinação e verifica todos os focos que os mesmos contêm. O algoritmo faz isso por meio de um *OR* bit a bit da combinação de focos de todos os vértices. Ao final, verifica-se a combinação final de focos para saber se todos estão presentes, retornando *True* caso se tenha todos os focos e *False*, caso contrário.

Algoritmo 4: HasAllFocuses

Data: $(combination, \mathbb{F}, \mathcal{R})$, where $combination$ is the combination of vertices, \mathbb{F} are the focuses and \mathcal{R} is the relationship between each vertex and its focuses

Result: a boolean indicating whether $combination$ has all focuses or not

```

1 begin
2   | focuses  $\leftarrow$  array of zeros of size  $F$ 
3   | foreach  $v \in combination$  do
4   |   | focuses  $\leftarrow$  focuses or  $\mathcal{R}(v)$ 
5   | end
6   | if focuses has all focuses  $\in \mathbb{F}$  then
7   |   | return True
8   | else
9   |   | return False
10  | end
11 end
```

Exercício 3

Analise as complexidades temporais e espaciais (usando notação assintótica) do algoritmo proposto no Exercício 2.

É possível compreender a ordem de complexidade temporal através de uma análise minuciosa de alguns pontos dos algoritmos apresentados no Exercício 2:

- Nas linhas (7) a (19) do algoritmo ZIKAZEROZ, são realizadas $\mathcal{O}(2^n)$ iterações, onde $n = |\mathbb{V}|$.
- Cada execução do procedimento GenerateCombination (Algoritmo 2) leva $\mathcal{O}(1)$.
- Cada execução do procedimento ExistsPath (Algoritmo 3) leva $\mathcal{O}(|\mathbb{V}|^2 + 2 \cdot |\mathbb{V}|) = \mathcal{O}(|\mathbb{V}|^2)$. O custo é similar ao custo de se realizar uma BFS em um grafo, considerando uma matriz de adjacência.
- Cada execução do procedimento HasAllFocuses (Algoritmo 4) custa $\mathcal{O}(|\mathbb{V}| \cdot |\mathbb{F}|)$, uma vez que é realizada uma operação de “logical OR” para o número de focos (tamanho $|\mathbb{F}|$) de cada vértice da combinação (tamanho máximo de $|\mathbb{V}|$).

Tendo em vista os dados anteriores, a complexidade temporal total do algoritmo é $\mathcal{O}(2^{|\mathbb{V}|} \cdot |\mathbb{V}|^3 \cdot |\mathbb{F}|)$, uma vez que cada um dos algoritmos – GenerateCombination, ExistsPath e HasAllFocuses – são executados para cada combinação gerada, devendo-se multiplicar as complexidades.

Como feito para a complexidade temporal, é possível compreender a ordem de complexidade espacial através de uma análise mais detalhada de algumas estruturas dos algoritmos:

- O uso de uma matriz de adjacência implica em um custo espacial de $\mathcal{O}(|\mathbb{V}|^2)$.
- O uso de uma matriz de focos, onde cada vértice apresenta um número de foco de tamanho $|\mathbb{F}|$, possui complexidade espacial de $\mathcal{O}(|\mathbb{V}| \cdot |\mathbb{F}|)$.
- O uso de uma matriz de melhores vértices (*bestVertices*) possui custo $\mathcal{O}(|\mathbb{V}|)$.
- A alocação do *array* de combinações possui custo de $\mathcal{O}(|\mathbb{V}|)$.
- O uso da fila pelo algoritmo ExistsPath possui custo de $\mathcal{O}(|\mathbb{V}|)$.
- O uso de um *array* de focos pelo algoritmo HasAllFocuses possui custo de $\mathcal{O}(|\mathbb{F}|)$.

A complexidade espacial total do algoritmo é $\mathcal{O}(|\mathbb{V}|^2 + 3|\mathbb{V}| + |\mathbb{V}| \cdot |\mathbb{F}| + |\mathbb{F}|)$. As complexidades espaciais foram somadas, porque o espaço é unicamente alocado para cada procedimento. Se $|\mathbb{V}| >>> |\mathbb{F}|$, podemos considerar a complexidade espacial do algoritmo como $\mathcal{O}(|\mathbb{V}|^2)$. Se temos $|\mathbb{F}| \geq |\mathbb{V}|^2$, o custo espacial pode aumentar para $\mathcal{O}(|\mathbb{V}|^3)$.

Exercício 4

Implemente o algoritmo proposto no Exercício 2 na linguagem de programação C, C++, JAVA ou PYTHON.

A implementação foi feita em PYTHON e se encontra na pasta “codigo” do presente trabalho.

Exercício 5

Execute testes da implementação do Exercício 4, propondo instâncias interessantes para o Problema ZikaZeroZ.

Foram realizados testes para as seguintes instâncias¹:

- **Conjunto 1:** grafos completos, nos quais todos os vértices estão conectados aos demais. Foram feitos testes com vértices $n = |\mathbb{V}| = \{5, 6, 7, \dots, 21\}$ e arestas $m = \frac{n \cdot (n-1)}{2}$, fixando-se o número de focos $r = |\mathbb{F}| = 4$.
- **Conjunto 2:** grafos com número de vértices fixo em $n = |\mathbb{V}| = 17$ e número de arestas fixo em $m = 136$, variando-se o número de focos, os quais assumiram os valores $r = |\mathbb{F}| = \{5, 25, 50, 100, 125, 150, 200, 250\}$.
- **Conjunto 3:** ciclo com número de vértices em $n = |\mathbb{V}| = \{5, 6, 7, \dots, 21\}$ e número de arestas $m = 136$, fixando-se o número de focos $r = |\mathbb{F}| = 4$.

Testes do Conjunto 1

Para os testes com grafos completos (Conjunto 1), o resultado da execução do algoritmo é apresentado na Tabela 1. Como pode-se observar, o tempo de execução aumenta à medida que se aumentam os vértices e arestas. Nota-se que esse aumento não é proporcional, ou seja, o tempo aumenta a uma taxa muito mais alta do que são adicionados mais vértices e arestas. Tais constatações podem ser também observadas nos gráficos das Figuras 2 e 3. Nessas figuras, pode-se notar que o tempo de execução possui um comportamento similar ao de uma função exponencial tanto ao se comparar com o número de vértices quanto de arestas do grafo. Tal característica justifica uma das limitações encontradas para a realização de mais testes para este conjunto: o número de instâncias foi limitado a 21 vértices devido ao tempo mais longo gasto para processar a resposta. Devido ao crescimento similar ao exponencial, ao aumentar o grafo de 20 vértices para 21 vértices, foi obtida uma diferença de cerca de 12 minutos no tempo de execução, dobrando-se o tempo para o grafo com $n = 20$. Por esse motivo, os testes foram feitos até uma instância de 21 vértices de um grafo completo.

Testes do Conjunto 2

Para o Conjunto 2 de testes, foram encontrados os resultados exibidos na Tabela 2, os quais também podem ser visualizados na Figura 4. Como pode-se observar, o tempo de execução aumenta à medida que se aumenta o número de focos existentes. Essa relação parece ser diretamente proporcional, se assemelhando a uma função linear, diferentemente do resultado visto para o Conjunto 1.

n	m	Tempo de Execução (s)
5	10	0.004021
6	15	0.009024
7	21	0.021011
8	28	0.048305
9	36	0.113366
10	45	0.241861
11	55	0.529237
12	66	1.189860
13	78	2.667712
14	91	6.145845
15	105	13.632422
16	120	28.858744
17	136	61.620125
18	153	129.654043
19	171	307.236836
20	190	616.869557
21	210	1307.023267

Tabela 1: Relação entre o número de vértices e arestas do grafo e o tempo de execução do algoritmo ZIKAZEROZ

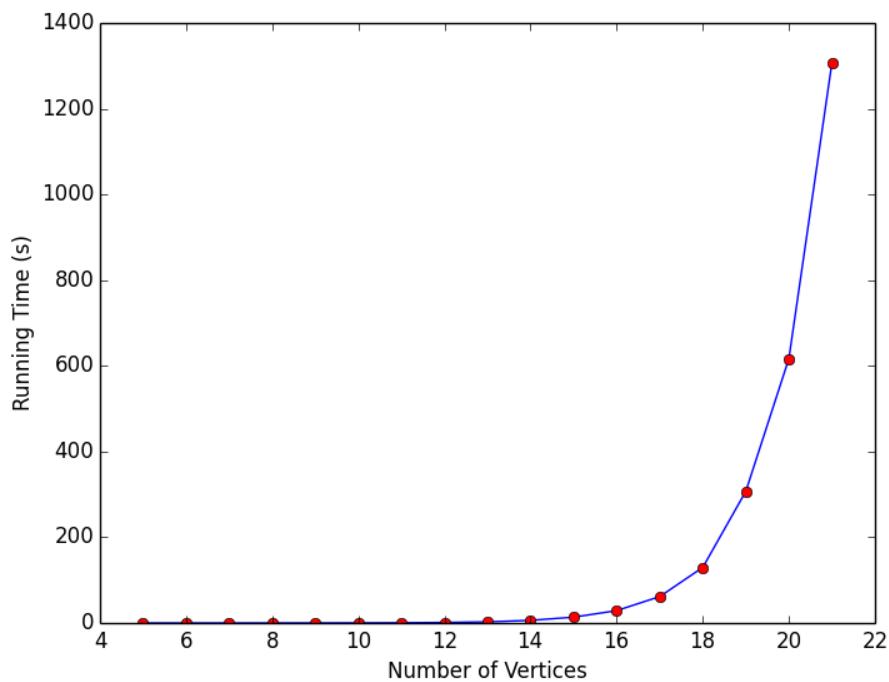


Figura 2: Relação entre Tempo de Execução e Número de Vértices para instâncias variando de 5 a 21
A linha azul apenas foi plotada para melhor visualização. Ela apenas conecta os dados obtidos (pontos do gráfico).

¹Todas as instâncias utilizadas nos testes se encontram na pasta “tests” do presente trabalho.

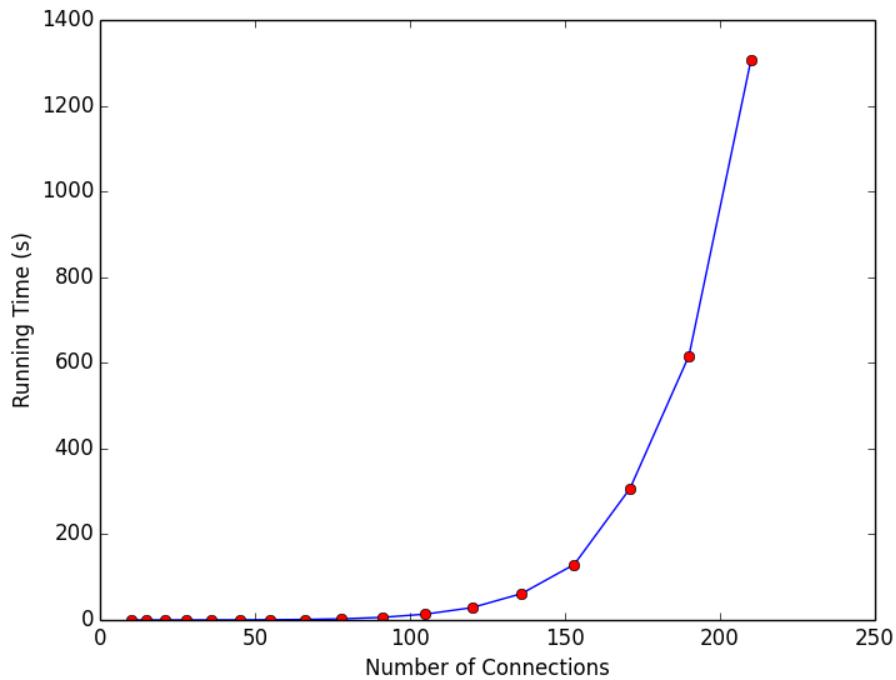


Figura 3: Relação entre Tempo de Execução e Número de Arestas

A linha azul apenas foi plotada para melhor visualização. Ela apenas conecta os dados obtidos (pontos do gráfico).

r	Tempo de Execução (s)
5	60.116136
25	64.099625
50	74.239282
100	94.963164
125	106.108643
150	118.519504
200	138.67890
250	155.986163
300	181.602287
350	202.008989

Tabela 2: Relação entre o número de focos de cada vértice e o tempo de execução do algoritmo ZIKAZEROZ

Testes do Conjunto 3

Com a finalidade de obter uma análise mais minuciosa para a relação entre o tempo de execução e o número de vértices e arestas, foram realizados os testes do Conjunto 3, os quais consideraram ciclos para os testes: aqui serão considerados grafos esparsos, em contraste com os grafos completos do Conjunto 1.

Para o Conjunto 3, o resultado da execução do algoritmo é apresentado na Tabela 3. Como pode-se observar, o tempo de execução aumenta à medida que se aumentam os vértices e arestas. Da mesma forma que para o Conjunto 1, nota-se que esse aumento não é proporcional, ou seja, o tempo aumenta a uma taxa muito mais alta do que são adicionados mais vértices e arestas. Tais constatações podem ser também observadas nos

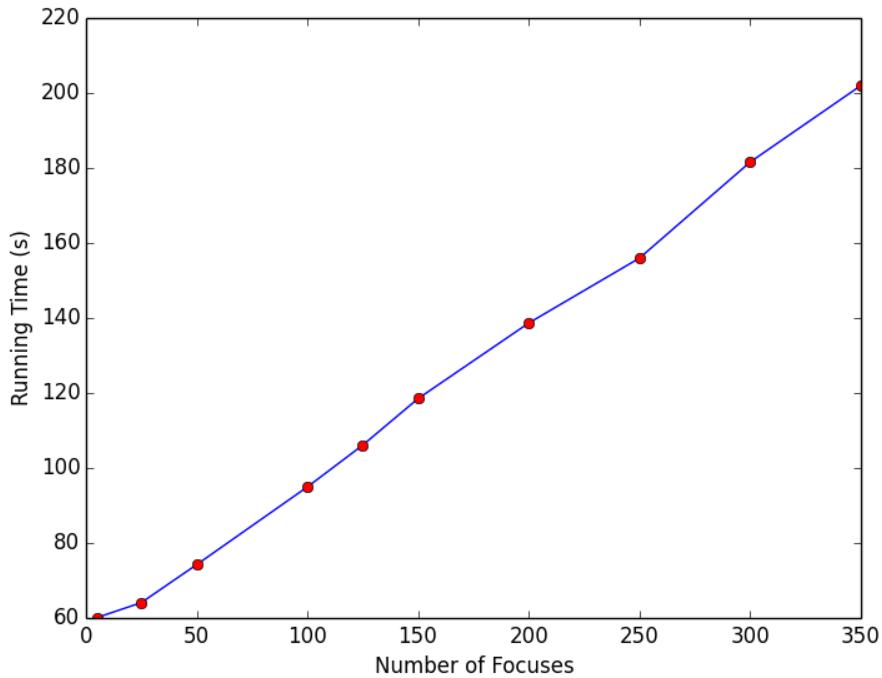


Figura 4: Relação entre Tempo de Execução e Número de Focos

A linha azul apenas foi plotada para melhor visualização. Ela apenas conecta os dados obtidos (pontos do gráfico).

gráficos das Figuras 2 e 3. Aqui também pode-se notar que o tempo de execução possui um comportamento similar ao de uma função exponencial tanto ao se comparar com o número de vértices quanto de arestas do grafo. Tal característica limitou a execução dos testes para o número de vértices de cada instância: foi possível rodar testes em grafos contendo, no máximo, 23 vértices. Como grafos de 23 vértices foram solucionados em um tempo de aproximadamente 30 minutos, este foi o limite até o qual foram executados os testes, uma vez que o tempo de execução dobrou a cada novo vértice adicionado.

Ao se estabelecer um paralelo, linha a linha, entre as tabelas do Conjunto 3 (Tabela 3) e Conjunto 1 (Tabela 1), percebe-se que existem diferenças. Isto provavelmente ocorre porque os grafos do Conjunto 3 são esparsos, enquanto os grafos do Conjunto 1 são grafos completos (portanto, densos). Dessa forma, os grafos do Conjunto 1 podem ser mais custosos para avaliação pelo procedimento **ExistsPath** (busca em largura) do algoritmo ZIKAZEROZ, uma vez que $m = |\mathbb{V}|^2$. Por outro lado, os grafos do Conjunto 3 possuem menor custo para o procedimento, sendo que $m = |\mathbb{V}|$.

n	m	Tempo de Execução (s)
5	5	0.003812
6	6	0.010540
7	7	0.014670
8	8	0.039008
9	9	0.069563
10	10	0.131652
11	11	0.293066
12	12	0.572023
13	13	1.234580
14	14	2.485918
15	15	5.281518
16	16	10.075940
17	17	21.805010
18	18	46.176279
19	19	110.963411
20	20	213.730517
21	21	446.501259
22	22	900.842778
23	23	1764.899504

Tabela 3: Relação entre o número de vértices e arestas do ciclo e o tempo de execução do algoritmo ZIKA-ZEROZ

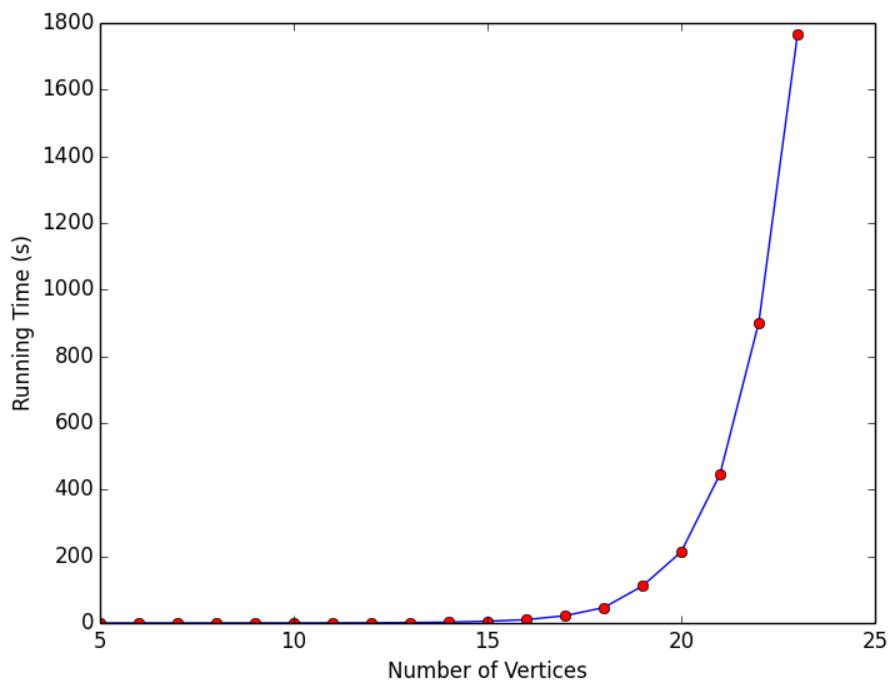


Figura 5: Relação entre Tempo de Execução e Número de Vértices para instâncias variando de 5 a 23
A linha azul apenas foi plotada para melhor visualização. Ela apenas conecta os dados obtidos (pontos do gráfico).

Exercício 6

Compare a análise e a execução, respectivamente, Exercícios 3 e 5. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

Conforme analisado para o algoritmo ZIKAZEROZ, a complexidade temporal é de $\mathcal{O}(2^{|V|} \cdot |V|^3 \cdot |F|)$. Pelo resultados experimentais vistos nas Figuras 3, 2 e 5, percebe-se que o algoritmo realmente apresenta um tempo de execução que parece crescer exponencialmente com o número de vértices do grafo ao fixar o número de focos. Tendo em vista que o custo computacional consiste em uma função exponencial multiplicada por funções polinomiais, ela “sobressai” às demais, ou seja, o que mostra que o resultado experimental se aproximou à previsão teórica neste caso. Analisando-se a Figura 4, onde se manteve o número de vértices e arestas e variou-se o número de focos, a complexidade temporal diz que o custo varia linearmente com o número de focos, uma vez que a função de complexidade temporal está multiplicada por $|F|$. Tal previsão teórica também é observada experimentalmente, o que pode ser corroborado pelos resultados mostrados na Figura 4. Nesta figura, mostra-se que o tempo de execução apresenta uma variação que parece crescer linearmente com o número de focos.

Aluno: João Francisco Neiva de Carvalho

Trabalho Prático - ZikaZeroZ

01/05/2016

Exercício 1

Dados do problema

1. Grafo $G(V, A)$, com $|V| = n$ e $|A| = m$, sendo V um conjunto de voluntários e A um conjunto de laços de amizade entre os voluntários de V .
2. Conjunto F , sendo $|F| = r$, sendo F o conjunto dos focos de reprodução do mosquito.
3. Relação $R(v)$: $V \rightarrow F$, definida para cada $v \in V$, explicitando os focos de reprodução aos quais o voluntário v tem acesso.

Objetivo

1. Determinar $V' \subset V$, tal que $G[V']$ seja conexo, todo foco $f \in F$ seja acessado por ao menos um voluntário $v \in V'$ e $|V'|$ seja a menor possível.

Modelagem

O próprio enunciado, considerando seus dados e objetivos, já indicou um caminho para a modelagem do problema. Seguiremos esse caminho, pois ele garante a obtenção de uma solução ótima, e buscaremos a solução de acordo com as seguintes etapas:

- ① Encontraremos todos os conjuntos V' que respeitem a seguinte condição:

$$V' \subset V \mid \forall f \in F \exists v \in V' \exists R(v): V \rightarrow F \quad (1)$$

Para cada um dos subconjuntos V' encontrados, faremos o seguinte: ② verificaremos se $G[V']$ é conexo. Se não for conexo, desconsideraremos V' como uma possível solução do problema. Se $G[V']$ for conexo, faremos o seguinte: ③ adicionaremos V' ao conjunto S de possíveis soluções do problema. Após termos determinado todos os elementos de S , ④ definiremos como solução o primeiro subconjunto V' que atenda à seguinte condição:

$$V' \subset S \mid |V'| = \min(|V'_i|) \forall V'_i \in S \quad (2)$$

Nesse processo de busca da solução, consideraremos as seguintes premissas:

1. $G(V, A)$ é um grafo simples, conexo ou desconexo.
2. $G(V, A)$ não é um dígrafo. Assim, $\forall (u, v) \in A, (u, v) = (v, u)$.
3. $R(v): V \rightarrow F$ cobre todos os elementos de F . Ou seja: $\forall f \in F \exists v \in V: \exists R(v): V \rightarrow F$. Falando de outra maneira: qualquer foco de reprodução está associado a, no mínimo, um voluntário. Ou ainda: se $R(v)$ fosse uma função, $R(v)$ seria sobrejetora.
4. Pela estrutura do arquivo de entrada, todo voluntário deverá acessar ao menos um foco de reprodução.

A seguir, apresentaremos os principais procedimentos que deverão ser utilizados para a realização das etapas **1** a **4** indicadas na modelagem.

Etapa **1 - Determinação de V' para a condição (1)**

Para determinarmos V' de acordo com a condição (1), faremos uma análise de $R(v)$, para determinarmos todas as possíveis combinações de voluntários que cobrem os elementos de F . Somente essas opções poderão ser uma solução do problema e, portanto, somente elas precisarão ser testadas.

A estrutura de dados para armazenar $R(v)$ bem como a forma de se realizar a análise de $R(v)$ são questões que deverão ser decididas no momento da definição do algoritmo. Por hora, basta observar que cada um dos subconjuntos V' será uma possível combinação de colaboradores de V capazes de cobrir todos os focos de reprodução indicados em F .

Etapa **2 - Verificando se $G[V']$ é conexo**

Para verificarmos se $G[V']$ (*subgrafo de G induzido por V'*) é conexo, deverá ser feita uma busca em profundidade em $G[V']$, a fim de encontrarmos uma árvore geradora desse subgrafo. Se tal árvore existir, seus vértices serão os mesmos de V' e, assim, $G[V']$ será conexo.

Etapas **3 e **4** - Definição da solução do problema**

Essas duas etapas poderão ser feitas separadamente, conforme indicado na modelagem, ou poderão ser combinadas em uma única etapa, para se otimizar o algoritmo.

Assim, poderemos encontrar todas as soluções possíveis para o problema, armazená-las em um conjunto e, posteriormente, verificar qual delas é de fato a solução do problema, por possuir o menor número de voluntários (*a menor cardinalidade*).

Ou então, poderemos também, a cada solução encontrada, verificar se ela é a melhor solução para o problema dentre as que já encontramos anteriormente, armazenando-a em caso positivo e descartando-a em caso negativo.

A definição da estratégia a ser adotada poderá ser tomada no momento da definição do algoritmo.

Exercício 2

Com base na modelagem proposta no Exercício 1, utilizaremos o seguinte algoritmo para resolver o problema:

```
1   Algoritmo ZikaZeroZ
2       S ← Ø      {Solução do problema}
3       para cada  $V'_i \subset V$ 
4           se  $G[V'_i]$  é conexo
5               se  $|V'_i| < (|V'| \in S)$  então
6                   S ←  $V'_i$ 
7       imprima S
```

A seguir, apresentaremos uma análise desse algoritmo.

Linhas 1 e 2

Declaração do algoritmo. S representa a solução do problema. Começamos atribuindo a S o conjunto vazio, uma vez que ainda não há solução.

Linha 3

Temos uma repetição para cada um dos subconjuntos $V'_i \subset V$ que satisfazem a condição (1) do exercício anterior. Para cada um desses subconjuntos, executaremos as linhas 4 a 6 do algoritmo.

Determinaremos V'_i manipulando a estrutura de dados que representará a relação $R(v)$. Essa estrutura será uma matriz $R_{n \times r}$, na qual as linhas representarão os voluntários e as colunas os focos de reprodução. Determinaremos o total de possíveis combinações de voluntários que garantem a cobertura de todos os focos. Cada uma dessas combinações será um subconjunto V'_i .

Para ilustrar esse processo, considere a matriz R da Figura 1. Ela corresponde àquela que será gerada pelo exemplo do enunciado do problema. As linhas da matriz representam os voluntários $v1$ a $v5$. As colunas representam os focos de reprodução $f1$ a $f4$.

	$f1$	$f2$	$f3$	$f4$
$v1$	1	1	0	0
$v2$	1	0	0	0
$v3$	0	1	0	0
$v4$	0	1	1	0
$v5$	0	0	0	1

Figura 1 – Matriz R para a relação $R(v)$ apresentada no exemplo do enunciado do problema.

O número de possíveis combinações de voluntários capazes de acessar todos os focos de reprodução é dado pelo produto da soma dos elementos das colunas dessa matriz. Assim, temos 6 possíveis combinações de voluntários ($6 = 2 \cdot 3 \cdot 1 \cdot 1$). Para determinarmos cada uma dessas combinações, ou seja, cada subconjunto V'_i , basta combinarmos cada um dos elementos com valor 1 na matriz. No exemplo, os subconjuntos V'_i serão:

$$\begin{aligned} V'_1 &= \{v1, v1, v4, v5\} & V'_3 &= \{v1, v4, v4, v5\} & V'_5 &= \{v2, v3, v4, v5\} \\ V'_2 &= \{v1, v3, v4, v5\} & V'_4 &= \{v2, v1, v4, v5\} & V'_6 &= \{v2, v4, v4, v5\} \end{aligned}$$

A cardinalidade máxima de cada V'_i será r (*o número total de focos de reprodução*). É importante notar que, como é possível que um mesmo voluntário acesse mais de um foco, um determinado subconjunto V'_i poderá ter cardinalidade menor do que a cardinalidade máxima prevista. Por isso é que V'_1 , V'_3 e V'_6 possuem cardinalidade 3 no exemplo que consideramos. A identificação dessa cardinalidade, antes da execução das próximas etapas do algoritmo, é importante, uma vez que um subconjunto V'_i será armazenado em S , como solução do problema, somente se a sua cardinalidade for menor do que a cardinalidade do conjunto V' já armazenado em S .

O número máximo de subconjuntos V'_i que encontraremos será n^r . Esse será o pior caso (*o que exigirá maior processamento*) e ocorrerá somente quando todos os voluntários puderem acessar todos os focos de reprodução.

A determinação dos subconjuntos V'_i da maneira proposta é factível, pois a matriz R deverá ser esparsa em situações reais. Ou seja, teremos um número de subconjuntos V'_i quase sempre muito menor do que n^r . No exemplo do enunciado do problema, temos 6 subconjuntos V' , ao passo que o número máximo desses subconjuntos é igual a 625 (5^4).

Apresentamos a seguir o algoritmo que utilizaremos para encontrar os subconjuntos V'_i .

```

1  Algoritmo SubConjuntoV'
2     $i \leftarrow r$    {começamos com o último foco de reprodução}
3    enquanto  $i > 0$ 
4       $p \leftarrow$  próximo voluntário para o foco  $f_i$  em  $R$ 
5      se existe  $p$ 
6         $V'_i \leftarrow V'_i$  anterior atualizado com  $p$  para  $f_i$ 
7         $i \leftarrow 0$    {O novo subconjunto  $V'_i$  foi encontrado}
8      senão
9         $V'_i \leftarrow V'_i$  anterior atualizado com o voluntário inicial para  $f_i$ 
10        $i \leftarrow i - 1$      {buscará um novo voluntário para o foco  $f_{i-1}$ }
11    retorna  $V'_i$ 
```

É interessante notar que, ao término desse algoritmo, se $V'_i = V_0$, onde V_0 é o primeiro subconjunto extraído de R , todas as possíveis combinações de voluntários, ou seja, todos os possíveis subconjuntos V'_i terão sido testados.

Linha 4

Conforme definido na modelagem apresentada no Exercício 1, verificaremos se $G[V'_i]$ é conexo realizando uma busca em profundidade em $G[V'_i]$. O grafo G será armazenado em uma Matriz de Adjacências G_{nxn} . Como a busca em profundidade será em um subgrafo de G , deveremos considerar em nosso algoritmo de busca apenas os vértices do subconjunto V'_i . A seguir, o algoritmo de busca que utilizaremos.

```

1  Algoritmo Busca (G: Grafo,  $V'_i$ : conjunto de vértices)
2    para cada vértice  $v$  de  $G$ 
3      marque  $v$  como não visitado
4     $V \leftarrow \emptyset$ 
5     $cardV \leftarrow 0$ 
6    para cada vértice  $v$  de  $G$ 
7      se  $v$  não foi visitado e  $v \in V'_i$ 
8        Busca-Prof( $v$ ,  $V'_i$ : conjunto de vértices)
9    Retorna  $V$ ,  $cardV$ 
```

```

1   Algoritmo Busca-Prof ( $v$ : vértice,  $V'_i$ : conjunto de vértices)
2       marque  $v$  como visitado
3        $V \leftarrow V \cup \{v\}$ 
4        $cardV \leftarrow cardV + 1$            {incrementando a cardinalidade de  $V$ }
5       para cada vértice  $w$  adjacente a  $v$ 
6           se  $w$  não foi visitado e  $w \in V'_i$ 
7               Busca-Prof( $w$ ,  $V'_i$ : conjunto de vértices)

```

Ao término desse algoritmo, um subconjunto de vértices V será retornado. Se $|V| = |V'_i|$, então $G[V'_i]$ será conexo e V'_i será uma possível solução do problema. Se não, $G[V'_i]$ não será conexo e V'_i deverá ser desconsiderado.

O algoritmo de busca também determina $|V|$, uma vez que esse valor será necessário na próxima linha do algoritmo principal (*linha 5*).

É interessante verificar que se G é desconexo, o algoritmo funcionará corretamente e uma solução existente em um único componente conexo de G poderá ser encontrada. Entretanto, caso V'_i envolva vértices de mais de um componente conexo de G , nenhuma solução poderá ser encontrada e V'_i não poderá ser uma solução do problema, como era de se esperar.

Linhas 5 e 6

Se $G[V'_i]$ é conexo, verificaremos se $|V'_i|$ é menor do que a cardinalidade do conjunto armazenado em S . Caso seja, atribuiremos a S o subconjunto V'_i , pois ele será a primeira possível solução do problema. Um valor inicial relacionado à cardinalidade do conjunto vazio deverá ser especificado na implementação do algoritmo, a fim de que a primeira solução encontrada possa ser considerada.

Nesse algoritmo, estamos considerando a melhoria sugerida no exercício anterior, ou seja, os passos **3** e **4** definidos na modelagem estão agrupados. Com isso, reduzimos o espaço e o tempo necessários à execução do algoritmo.

Linha 7

Após o processamento de todos os subconjuntos V'_i , imprimimos S , que contém a solução do problema, ou o conjunto vazio, se nenhuma solução tiver sido encontrada.

É importante verificar que o algoritmo tem execução finita, ou seja, o algoritmo pára após um número finito de operações. De fato, isso ocorre, uma vez que a repetição da linha 3 será realizada um número finito de vezes e, no máximo, n' vezes. Além disso, o algoritmo de busca em profundidade que utilizamos é convencional e possui término de execução garantido.

A seguir, apresentamos o *Algoritmo ZikaZeroZ* de forma mais detalhada, mais próxima de uma implementação, considerando suas principais estruturas de dados e suas principais instruções.

```

1   Algoritmo ZikaZeroZ
2       declare  $n, m, r, i, menor$  numérico
3       declare  $R_{nxr}, G_{nxn}, S_{1xn}, V_{0\cdot 1xn}, V_{1xn}, V_{i1xn}$  numérico
4       leia arquivo de entrada com  $n, m, G, r$  e  $R$ 
5        $i \leftarrow \prod_{j=1}^r (\sum_{i=1}^n r_{ij})$ 
6        $menor \leftarrow -1$ 
7        $S \leftarrow \emptyset$ 
8        $V_0 \leftarrow SubConjunto(0)$            {Define  $V_0$  manipulando a matriz  $R$ }
9        $V_i \leftarrow V_0$ 
10      repita enquanto  $i > 0$ 
11           $V \leftarrow Busca(G[V_i])$            {Busca em profundidade}
12          se  $|V| = |V_i|$  então           { $G[V_i]$  é conexo e  $V_i$  é uma possível solução}
13              se ( $menor = -1$ ) ou ( $|V_i| < menor$ ) então           { $V_i$  é a solução}
14                   $menor \leftarrow |V_i|$ 
15                   $S \leftarrow V_i$ 
16          fim se
17          fim se
18           $i \leftarrow i - 1$ 
19           $V_i \leftarrow SubConjunto(i)$            {Define  $V_i$  manipulando a matriz  $R$ }
20      fim repita
21      imprimir  $S$ 
22  fim algoritmo

```

Exercício 3

Para determinarmos as complexidades temporais e espaciais, repetiremos a seguir o algoritmo do Exercício 2.

Análise da Complexidade Temporal

<u>Instrução</u>	<u>Custo</u>	<u>Vezes</u>
1 <u>Algoritmo</u> <i>ZikaZeroZ</i>	0	1
2 <u>declare</u> <i>n, m, r, i, menor numérico</i>	c_1	1
3 <u>declare</u> <i>R_{nxr}, G_{nxn}, S_{1xn}, V_{01xr}, V_{1xn}, V_{i1xr} numérico</i>	c_2	1
4 <u>leia</u> arquivo de entrada com <i>n, m, G, r e R</i>	c_3	t_1
5 $i \leftarrow \prod_{j=1}^r (\sum_{i=1}^n r_{ij})$	c_4	1
6 <i>menor</i> $\leftarrow -1$	c_5	1
7 $S \leftarrow \emptyset$	c_6	1
8 $V_0 \leftarrow SubConjuntoV_0$	c_7	1
9 $V_i \leftarrow V_0$	c_8	1
10 <u>repita enquanto</u> <i>i > 0</i>	c_9	$1 + \prod_{j=1}^r (\sum_{i=1}^n r_{ij})$
11 $V \leftarrow Busca(G[V_i])$	c_{10}	$\prod_{j=1}^r (\sum_{i=1}^n r_{ij})$
12 <u>se</u> $ V = V_i $ <u>então</u>	c_{11}	$\prod_{j=1}^r (\sum_{i=1}^n r_{ij})$
13 <u>se</u> (<i>menor</i> = -1) ou ($ V_i < menor$) <u>então</u>	c_{12}	t_2
14 <i>menor</i> $\leftarrow V_i $	c_{13}	t_3
15 $S \leftarrow V_i$	c_{14}	t_3
16 <u>fim se</u>	0	t_2
17 <u>fim se</u>	0	$\prod_{j=1}^r (\sum_{i=1}^n r_{ij})$
18 $i \leftarrow i - 1$	c_{15}	$\prod_{j=1}^r (\sum_{i=1}^n r_{ij})$
19 $V_i \leftarrow SubConjunto(i)$	c_{16}	$\prod_{j=1}^r (\sum_{i=1}^n r_{ij})$
20 <u>fim repita</u>	0	$\prod_{j=1}^r (\sum_{i=1}^n r_{ij})$
21 <u>imprimir</u> <i>S</i>	c_{17}	1
22 <u>fim algoritmo</u>	0	1

Observação: Uma instrução adicional para alocação de memória poderia ter sido introduzida logo após a linha 4. Entretanto, estamos considerando que o processo de alocação de memória tem custo constante independentemente da quantidade de memória a ser alocada.

Para calcularmos o tempo de execução, $T(n, m, r)$, do *Algoritmo ZikaZeroZ*, basta somarmos os produtos das colunas *Custo* e *Vezes* de cada instrução, obtendo o seguinte resultado:

$$T(n, m, r) = (c_1 + c_2 + c_4 + c_5 + c_6 + c_8 + c_9) + (c_9 + c_{10} + c_{11} + c_{15} + c_{16}) \prod_{j=1}^r (\sum_{i=1}^n r_{ij}) + c_3 t_1 + c_{12} t_2 + (c_{13} + c_{14}) t_3 + c_7 + c_{17} \quad (1)$$

Determinação dos custos

Os custos c_1, c_6, c_8 e c_9, c_{12} a c_{15} são constantes, dependendo exclusivamente das características da máquina, pois eles se referem somente a atribuições e comparações.

O custo c_{17} depende do tamanho de S , pois cada um dos elementos de S será impresso. Assim, temos que:

$$c_{17} = O(n). \quad (2)$$

O custo c_{11} depende do tamanho de V_i , pois a cardinalidade de V será determinada durante a busca em profundidade em $G[V_i]$. Para determinarmos a cardinalidade de V_i verificaremos cada um dos elementos de V_i uma única vez. Assim, temos que:

$$c_{11} = O(r). \quad (3)$$

O custo c_{10} é o custo do algoritmo de busca em profundidade em um grafo. Sabemos que tal custo possui a função $g(n, m) = n + m$ como limite assintótico superior. É importante observar que, apesar da estarmos verificando se $G[V_i]$ é conexo, a busca será realizada em G ; assim, n e m serão as quantidades de vértices e arestas de G . Além disso, consideramos que podemos verificar se um determinado vértice de G pertence a em V_i em um tempo constante. Desse modo, temos que:

$$c_{10} = O(n + m) \quad (4)$$

O custo c_7 depende do tamanho de R . Uma vez que a identificação do primeiro subconjunto V_i , o subconjunto inicial V_0 , exigirá uma busca em toda matriz R , temos que:

$$c_7 = O(nr) \quad (5)$$

Já o custo c_{16} está associado ao algoritmo que encontrará os subconjuntos V_i . Apresentamos esse algoritmo no Exercício 2. A seguir, rerepresentaremos esse algoritmo de forma mais detalhada, a fim de estimarmos o seu tempo de execução (T_{SubV}), que será o custo c_{16} .

Algoritmo SubConjuntoV' (R: matriz com a relação R(v), r: total de focos de reprodução, n: total de voluntários, V: conjunto de voluntários)

<i>Instrução</i>	<i>Custo</i>	<i>Vezes</i>
$f \leftarrow r$	k_1	1
$p \leftarrow -1$	k_2	1
enquanto $p = -1$ e $f > 0$	k_3	t_1
$v \leftarrow V[f] + 1$	k_4	$t_1 - 1$
enquanto $p = -1$ e $v \leq n$	k_5	t_2
se $R[v, f] = 1$	k_6	t_3
$p \leftarrow v$	k_7	t_4
$v \leftarrow v + 1$	k_8	t_3
se $p \neq -1$	k_9	$t_1 - 1$
$V[f] \leftarrow p$	k_{10}	t_5
senão	k_{11}	t_6
$V[f] \leftarrow$ voluntário inicial para f	k_{12}	t_6
$f \leftarrow f - 1$	k_{13}	$t_1 - 1$

É fácil ver que os custos k_1 a k_{13} são constantes que dependem exclusivamente da máquina. Já os tempos t_1 a t_6 dependem da configuração da matriz R . Como buscamos um limite assintótico para

o tempo de execução do algoritmo, consideraremos a configuração da matriz R no melhor e no pior caso. Como R é uma matriz $n \times r$, consideraremos, em ambos os casos, que R é uma matriz quadrada de ordem w sendo $w = \max(n, r)$. Faremos isso para facilitar as análises, gerando uma relação entre os tempos das estruturas de repetição dos algoritmos.

Analizando o algoritmo superficialmente, vemos que a complexidade será nr no pior caso. Assim, calcularemos a complexidade utilizando a consideração que mencionamos e, caso ela seja mais fraca (*menos restrita*) do que a complexidade nr , consideraremos nr a complexidade do algoritmo para esse caso.

No pior caso para esse algoritmo, teremos um voluntário por foco de reprodução. Assim, faremos R ser uma matriz identidade de ordem w , uma vez que, pela premissa 4 indicada no Exercício 1, R não pode ter somente a primeira linha com valores 1 e todas as demais com valores 0, caso que, de fato, seria o pior caso se pudesse existir.

Dessa forma, no pior caso, a única combinação possível de voluntários é a combinação inicial, isto é, sempre existirá um único subconjunto V , que será o próprio conjunto de vértices do grafo $G(V, A)$, capaz de solucionar o problema. Além disso, as condições $p = -1$ sempre serão verdadeiras nas duas estruturas de repetição, de modo que essas repetições serão finalizadas apenas quando toda a matriz R tiver sido processada. Assim, temos os seguintes tempos:

$$t_1 = w + 1$$

$$t_2 = \sum_{i=1}^w (w + 1 - i) = w^2 + w - \sum_{i=1}^w i = \frac{w(w+1)}{2}$$

$$t_3 = \sum_{i=1}^w (w + 1 - i - 1) = \sum_{i=1}^w (w - i) = w^2 - \sum_{i=1}^w i = \frac{w(w-1)}{2}$$

$$t_4 = t_5 = 0$$

$$t_6 = t_1 - 1 = w$$

$$\begin{aligned} \therefore T_{SubV} &= (k_1 + k_2 + k_3) + (k_3 + k_4 + k_9 + k_{11} + k_{12} + k_{13})w \\ &\quad + k_5 \frac{w(w+1)}{2} + (k_6 + k_8) \frac{w(w-1)}{2} \end{aligned}$$

$$\therefore T_{SubV} = O(w^2) = O(\max(n, r)^2)$$

Como $\max(n, r)^2$ é mais fraco do que nr , conforme explicamos anteriormente, consideraremos:

$$c_{16} = O(nr) \tag{6}$$

No melhor caso, R será uma matriz com todos os seus elementos iguais a 1, significando que todos os voluntários poderão acessar todos os focos de reprodução.

É interessante notar que, para o algoritmo principal, *Algoritmo ZikaZeroZ*, esse será o pior caso, pois, embora qualquer um dos voluntários isoladamente seja a solução do problema, o algoritmo não identificará esse caso particular de forma imediata. Ele precisará fazer todas as operações para identificá-lo, percorrendo toda a matriz R inúmeras vezes, testando todos os subconjuntos V_i .

No caso desse algoritmo, é importante notar que para qualquer subconjunto V que seja um vértice anterior ao último vértice do grafo, o algoritmo terá um processamento mínimo. Isto é, todas as instruções serão executadas uma única vez, exceto as instruções de custo k_3 e k_5 , que serão executadas duas vezes.

O número maior de execuções ocorrerá quando o subconjunto V for o último vértice do grafo, uma vez que, nesse caso, todas as posições do vetor V voltarão à posição inicial. Nessa situação, as condições $p = -1$ também sempre serão verdadeiras nas duas estruturas de repetição, de modo que ambas serão finalizadas apenas quando toda a matriz R tiver sido processada.

Como essa situação é um limite superior para todas as outras, consideraremos que os tempos associados a ela serão os tempos do algoritmo no melhor caso. Assim, temos que:

$$\begin{aligned}t_1 &= w + 1 \\t_2 &= \sum_{i=1}^w 1 = w \\t_3 &= t_4 = t_5 = 0 \\t_6 &= t_1 - 1 = w\end{aligned}$$

$$\begin{aligned}\therefore T_{SubV} &= (k_1 + k_2 + k_3) + (k_3 + k_4 + k_5 + k_9 + k_{11} + k_{12} + k_{13})w \\\therefore c_{16} &= O(w) = O(\max(n, r))\end{aligned}\tag{7}$$

Determinação dos tempos

O tempo t_1 está relacionado ao tamanho do arquivo de entrada que especifica G e $R(v)$. Como estamos trabalhando com uma *matriz de adjacências*, seja qual for o tipo do grafo G , o número de leituras no arquivo sempre será o número de arestas de G . Considerando a relação $R(v)$, temos que distinguir o melhor e o pior caso. No melhor caso, cada um dos focos de reprodução estará relacionado a apenas um voluntário. No pior caso, cada um dos voluntários poderá acessar todos os focos de reprodução. Sendo assim, temos:

$$\text{Melhor caso: } t_1 = 2 + m + 1 + n \tag{8}$$

$$\text{Pior caso: } t_1 = 2 + m + 1 + nr \tag{9}$$

Já para as instruções associadas aos tempos t_2 e t_3 , podemos considerar que, no melhor caso, elas serão executadas uma única vez. No pior caso, elas serão executadas todas as vezes que a instrução da linha 11 for executada. Assim, temos que:

$$\text{Melhor caso: } t_2 = t_3 = 1 \tag{10}$$

$$\text{Pior caso: } t_2 = t_3 = \prod_{j=1}^r (\sum_{i=1}^n r_{ij}) \tag{11}$$

Tempo de execução do Algoritmo ZikaZeroZ

Para determinarmos o tempo de execução do algoritmo, precisamos considerar que, no melhor caso, teremos um único voluntário por foco de reprodução e que, no pior caso, cada voluntário estará associado a todos os focos de reprodução, ou seja, $r_{ij} = 1 \forall i, j$. Assim, temos que:

$$\text{Melhor caso: } \prod_{j=1}^r (\sum_{i=1}^n r_{ij}) = 1 \tag{12}$$

$$\text{Pior caso: } \prod_{j=1}^r (\sum_{i=1}^n r_{ij}) = n^r \tag{13}$$

Agora, levando (12) a (13) em (1), determinaremos o tempo de execução do *Algoritmo ZikaZeroZ* para o melhor e para o pior caso.

Melhor caso:

$$\begin{aligned} T(n, m, r) &= (c_1 + c_2 + c_4 + c_5 + c_6 + c_8 + c_9) \\ &+ (c_9 + c_{10} + c_{11} + c_{15} + c_{16}) \prod_{j=1}^r \left(\sum_{i=1}^n r_{ij} \right) \\ &+ c_3 t_1 + c_{12} t_2 + (c_{13} + c_{14}) t_3 + c_7 + c_{17} \end{aligned}$$

$$\begin{aligned} T(n, m, r) &= (c_1 + c_2 + c_4 + c_5 + c_6 + c_8 + c_9) \\ &+ (c_9 + O(n+m) + O(r) + c_{15} + O(nr)) \cdot 1 \\ &+ c_3(2+m+1+n) + c_{12} \cdot 1 + (c_{13} + c_{14}) \cdot 1 + O(nr) + O(n) \end{aligned}$$

$$\therefore T_{\text{Melhor}}(n, m, r) = O(nr + m) \quad (14)$$

Pior caso:

$$\begin{aligned} T(n, m, r) &= (c_1 + c_2 + c_4 + c_5 + c_6 + c_8 + c_9) \\ &+ (c_9 + c_{10} + c_{11} + c_{15} + c_{16}) \prod_{j=1}^r \left(\sum_{i=1}^n r_{ij} \right) \\ &+ c_3 t_1 + c_{12} t_2 + (c_{13} + c_{14}) t_3 + c_7 + c_{17} \end{aligned}$$

$$\begin{aligned} T(n, m, r) &= (c_1 + c_2 + c_4 + c_5 + c_6 + c_8 + c_9) \\ &+ (c_9 + O(n+m) + O(r) + c_{15} + O(\max(n, r))) \cdot n^r \\ &+ c_3(2+m+1+nr) + c_{12} \cdot n^r + (c_{13} + c_{14}) \cdot n^r + O(nr) + O(n) \end{aligned}$$

$$\begin{aligned} T(n, m, r) &< (c_1 + c_2 + c_4 + c_5 + c_6 + c_8 + c_9) \\ &+ c_9 n^r + k_1 n n^r + k_1 m n^r + k_2 r n^r + c_{15} n^r + k_3 \max(n, r) n^r \\ &+ c_3(2+m+1+nr) + c_{12} n^r + (c_{13} + c_{14}) n^r + k_4 nr + k_5 n \end{aligned}$$

sendo k_1 a k_5 constantes positivas. Como m , no pior caso, pode ser igual a $\frac{n(n-1)}{2}$, temos:

$$\therefore T_{\text{Pior}}(n, m, r) = O(n^{r+2}) \quad (15)$$

É importante registrar que verificamos possibilidades mais restritas para o limite assintótico superior, como, por exemplo, $n^{r+1} + m + nr$. Contudo, os cálculos que fizemos indicaram que a razão de $T(n)$ sobre essas outras possíveis funções sempre crescia com o aumento de n, m e r , indicando que a função proposta não poderia ser um limite assintótico superior.

É interessante salientar que, em situações reais, provavelmente, estaremos bem mais próximos do melhor caso do que do pior caso, pois, como dissemos anteriormente, acreditamos que R deve

ser uma matriz esparsa na maioria das situações. Acreditamos que, dificilmente, teremos uma matriz R totalmente preenchida com elementos de valor 1, uma vez que isso significa que todos os voluntários teriam acesso a todos os focos de reprodução.

Análise da Complexidade Espacial

Analizando o algoritmo, vemos que as principais estruturas de dados, aquelas que dependem do tamanho da entrada do algoritmo, são as seguintes: R_{nxr} , G_{nxn} , S_{1xn} , V_{01xr} , V_{1xn} e V_{l1xr} . Assim, temos que a quantidade de espaço (*quantidade de memória*) necessária ao algoritmo é a seguinte:

$$Q(n, r) = nr + n^2 + n + r + n + r = n^2 + nr + 2(n + r)$$

Sendo assim, a complexidade espacial do algoritmo, mesmo para $r \gg n$, como demonstraram os cálculos que fizemos, pode ser dada por:

$$\therefore Q(n, r) = O(n^2 + nr) \quad (16)$$

As complexidades solicitadas no exercício estão apresentadas em (14), (15) e (16).

É interessante notar que, em casos reais, é razoável supor $n \cong r$. Nesses casos, para n e r muito grandes, as igualdades (14), (15) e (16) poderiam ser expressas por:

$$T_{Melhor}(n, m, r) = O(n^2) \quad T_{Pior}(n, m, r) = O(n^{r+2}) \quad Q(n, r) = O(n^2)$$

Exercício 4

Implementamos o algoritmo utilizando a linguagem de programação **C**. Criamos um único arquivo com todas as funções necessárias. Chamamos esse arquivo de **zikazeroz.c**.

Os protótipos das funções criadas são os seguintes:

```
int      main(int argc,char *argv[]);
void    zikazeroz(char* instancia);
int     digitos(int);
void    subConjuntoVInicial(int** R, int* VInicial, int n, int r);
void    subConjuntoVLinha(int** R, int* VLinha, int* VInicial, int n, int r);
int     busca(int** G, int* V, int n, int* VLinha, int r);
void    busca_prof(int** G, int* V, int n, int* VLinha, int r, int vertice, int* card);
int     pertence(int vertice, int* vertices, int n);
int     cardinalidade(int* vetor, int d, int n);
```

Desenvolvemos e testamos o programa em uma máquina com sistema operacional Windows 10, utilizando o compilador LCC, disponível em <https://www.cs.virginia.edu/~lcc-win32/>.

Também compilamos e testamos o programa nas máquinas Linux do laboratório do DCC. Nessas máquinas, a compilação do programa pode ser feita com um dos seguintes comandos:

```
gcc zikazeroz.c -std=c99 -o zikazeroz    ou
./compilar.sh
```

Já a execução pode ser feita com um dos seguintes comandos:

```
./zikazeroz    ou
./executar.sh
```

Os arquivos *bash (shell script)* **compilar.sh** e **executar.sh** estão sendo fornecidos.

Entrada do Programa, Instâncias do Algoritmo e Parâmetros do Programa

Um arquivo de entrada para o programa é uma instância do algoritmo. Fizemos o programa de forma que diferentes instâncias possam ser definidas em arquivos de entrada diferentes.

Os nomes dos arquivos de entrada devem seguir o padrão **ZikaZeroZn.txt**, onde **n** é um número natural ($n \in \{0, 1, 2, 3, 4, \dots\}$) que identifica a instância. O arquivo **ZikaZeroZ.txt** também é um arquivo de instância. Nós o reservamos para a instância do exemplo do enunciado do problema.

Para a execução de uma instância específica, precisamos apenas passar o número da instância como um parâmetro para o programa, utilizando comandos de execução com a seguinte forma: **./zikazeroz n** ou **./executar.sh n**, onde **n** é o número da instância.

A seguir, alguns exemplos para a execução do programa:

./zikazeroz ou ./executar.sh	Executa a instância do arquivo ZikaZeroZ.txt . Instância do exemplo do enunciado do problema.
./zikazeroz 0 ou ./executar.sh 0	Executa a instância do arquivo ZikaZeroZ0.txt .

./zikazeroz 1 ou ./executar.sh 1 Executa a instância do arquivo **ZikaZeroZ1.txt**

Todos os arquivos de instância devem ser do tipo texto e todos eles devem estar localizados no mesmo diretório do arquivo executável do programa. Caso o programa não encontre um arquivo de instância (*arquivo de entrada*), uma mensagem de erro será exibida.

Saída do Programa e Opções de Debug

Após a execução do programa, conforme especificado, uma única linha é impressa, contendo os voluntários selecionados como solução do problema, em ordem crescente, separados por um espaço. Caso a instância não tenha solução, uma mensagem com essa informação é apresentada.

Opções de *debug*, que permitem a impressão de mensagens relacionadas aos passos de execução do programa, também estão disponíveis. Contudo, tais opções são alteráveis apenas em código. Desse modo, para alterá-las, uma nova compilação do programa precisa ser realizada.

Exercício 5

Para testarmos a implementação do algoritmo, criamos os seguintes quatro grupos de instâncias. O primeiro deles contendo 5 instâncias e os demais contendo 4 instâncias cada:

- Grupo 1: instâncias básicas.
- Grupo 2: instâncias com aumentos progressivos de n e m , com r constante.
- Grupo 3: instâncias com aumentos progressivos de r , com n e m constantes.
- Grupo 4: instâncias com aumentos progressivos de n , m e r .

Para cada uma das 17 instâncias de teste, consideramos os valores de n , m , r e os valores de duas outras grandezas: s que é o total de combinações verificadas pelo algoritmo e que nos dá uma noção de quanto esparsa é a matriz R ; e t , que é o tempo de execução do programa.

Nas instâncias dos grupos 2 a 4, procuramos manter constante o total de combinações de cada instância, uma vez que já está claro que, quanto maior for o número de combinações, maior será o tempo de execução do algoritmo, e o nosso objetivo nesses grupos era determinar a influência das funções do programa (*partes do algoritmo*) no seu tempo de execução.

Todos os testes, cujos resultados apresentamos aqui, foram realizados em uma máquina com a seguinte configuração:

- Processador Intel® Core™ i5-4210U 1,70GHz a 2,40GHz.
- Memória RAM 4,0GB.
- Sistema operacional de 64 bits, processador com base em x64.
- Windows 10 Home Single Language.

O executável do programa foi gerado com o compilador LCC, disponível em <https://www.cs.virginia.edu/~lcc-win32/>.

Com relação ao tempo de processamento, a observação mais relevante que fizemos em função dos testes foi a seguinte: o aumento do tempo de processamento é mais acentuado em função do aumento do número de focos de reprodução (*aumento de r*) do que em função do aumento do número de voluntários e dos seus laços de amizade (*aumento do grafo*). Os testes dos grupos 1 e 2 comprovam isso.

Tal observação está de acordo com as nossas expectativas, uma vez que o aumento de r implica o maior processamento das funções menos otimizadas ou mais custosas do algoritmo: as funções que manipulam a matriz R , a função que determina a cardinalidade de um vetor e a que verifica se um determinado elemento pertence a um vetor.

Outra observação importante é que o algoritmo não é viável para trabalhar com grandes volumes de dados. Os testes do grupo 4 mostram que, com volumes de dados um pouco maiores, o tempo de execução do programa aumenta consideravelmente (*ver o teste da instância ZikaZeroZ43.txt*). Mesmo com um número reduzido de repetições, é grande o tempo de processamento em função do volume de dados.

A seguir, apresentaremos cada um dos grupos e cada uma das instâncias de teste que utilizamos. Estamos fornecendo, juntamente com o código fonte, os arquivos dessas instâncias.

Instâncias do Grupo 1: instâncias básicas

Objetivamos neste grupo testar o funcionamento do programa em relação à correção da implementação do algoritmo, ao critério de escolha da solução e ao trabalho com grafos conexos, desconexos e completos.

ZikaZeroZ.txt

Instância apresentada como exemplo do enunciado do problema. Possui solução.
 $n = 5, m = 6, r = 4, s = 6, t = 10$ ms.

ZikaZeroZ10.txt

Instância com um grafo desconexo, com 5 vértices isolados, mas com cada voluntário acessando todos os focos de reprodução. Possui solução.
 $n = 5, m = 0, r = 10, s = 9.765.625, t = 1.810$ ms.

ZikaZeroZ11.txt

Instância com um grafo desconexo, contendo duas componentes conexas, sendo que os voluntários da segunda componente acessam todos os focos. Possui solução.
 $n = 5, m = 3, r = 10, s = 10.368, t = 10$ ms.

ZikaZeroZ12.txt

Instância com um grafo desconexo, contendo duas componentes conexas, sendo que os voluntários de nenhuma dessas componentes têm acesso a todos os focos. Não possui solução.

$n = 5, m = 3, r = 10, s = 4.608, t = 10$ ms.

ZikaZeroZ13.txt

Instância com um grafo completo e com todos os voluntários acessando todos os focos de reprodução. Possui solução.
 $n = 5, m = 10, r = 10, s = 9.765.625, t = 3.290$ ms.

Instâncias do Grupo 2: instâncias com aumentos progressivos de n e m , com r constante

Desejamos medir a influência do tamanho do grafo no tempo de execução do programa. Quanto maior o grafo, maior será o tempo de processamento das funções relacionadas à busca em profundidade.

ZikaZeroZ20.txt

Possui solução. $n = 50, m = 50, r = 5, s = 1.080.450, t = 490$ ms.

ZikaZeroZ21.txt

Possui solução. $n = 100, m = 100, r = 5, s = 1.069.200, t = 730$ ms.

ZikaZeroZ22.txt

Possui solução. $n = 200, m = 200, r = 5, s = 1.114.400, t = 1.320$ ms.

ZikaZeroZ23.txt

Possui solução. $n = 1.000, m = 1.000, r = 5, s = 1.080.000, t = 4.080$ ms.



Figura 2 – Gráfico das instâncias do grupo 2.

Instâncias do Grupo 3: instâncias com aumentos progressivos de r, com n e m constantes

Desejamos medir a influência do total de focos de reprodução no tempo de execução do programa. Quanto maior for o número de focos, maior será o tempo de processamento das funções relacionadas à determinação da cardinalidade e dos subconjuntos V' .

ZikaZeroZ30.txt

Possui solução. $n = 5$, $m = 6$, $r = 50$, $s = 1.048.576$, $t = 890$ ms.

ZikaZeroZ31.txt

Possui solução. $n = 5$, $m = 6$, $r = 100$, $s = 1.048.576$, $t = 1.680$ ms.

ZikaZeroZ32.txt

Possui solução. $n = 5$, $m = 6$, $r = 200$, $s = 1.048.576$, $t = 3.250$ ms.

ZikaZeroZ33.txt

Possui solução. $n = 5$, $m = 6$, $r = 1.000$, $s = 1.048.576$, $t = 15.910$ ms.



Figura 3 – Gráfico das instâncias do grupo 3.

Instâncias do Grupo 4: instâncias com aumentos progressivos de n , m e r

Desejamos medir a influência do volume de dados no tempo de execução do programa. Quanto maior for o volume de dados, maior será o tempo de processamento. Queremos verificar a viabilidade de processar grandes volumes de dados, identificando um volume máximo a partir do qual o processamento se torna inviável.

ZikaZeroZ40.txt

Possui solução. $n = 50$, $m = 50$, $r = 50$, $s = 262.144$, $t = 2.830$ ms.

ZikaZeroZ41.txt

Possui solução. $n = 100$, $m = 100$, $r = 100$, $s = 262.144$, $t = 12.410$ ms.

ZikaZeroZ42.txt

Possui solução. $n = 200$, $m = 200$, $r = 200$, $s = 262.144$, $t = 53.390$ ms.

ZikaZeroZ43.txt

Possui solução. $n = 1000$, $m = 1000$, $r = 1000$, $s = 262.144$, $t = 1.733.780$ ms.



Figura 4 – Gráfico das instâncias do grupo 4.

É possível notar, pelos gráficos apresentados, que, de modo geral, o tempo de processamento é uma função exponencial da entrada. Analisando os gráficos dos grupos 2 e 3, é possível verificar que aumentos de r fazem o tempo de processamento crescer mais do que aumentos de n e m .

Analizando o gráfico do grupo 4, verificamos que o tempo de processamento também cresce exponencialmente com o volume total de dados da instância. O processamento da instância *ZikaZeroZ43*, a maior instância que testamos, levou cerca de 30 minutos para ser concluído, ao passo que a instância *ZikaZeroZ42*, que é 80% menor, teve um tempo de processamento 97% menor. Portanto, há de se definir qual é o limite de tempo que se considerará aceitável para o processamento de uma determinada instância. O algoritmo produz soluções ótimas, mas não é eficiente para todas as instâncias.

Finalmente, é importante considerar que apenas instâncias pequenas podem ter uma matriz R totalmente densa (*totalmente preenchida com valores 1*). Como regra, quanto mais densa for a matriz R , maior será o tempo de processamento do algoritmo. Instâncias grandes com matrizes R densas possuem tempo de processamento inviável.

Exercício 6

Pelos testes que realizamos, acreditamos que as previsões teóricas, apresentadas no Exercício 3, estão de acordo com os resultados experimentais obtidos no Exercício 5.

Em uma comparação qualitativa, é possível verificar essa concordância. Nossos testes indicaram que aumentos em r elevam mais o tempo de processamento do que aumentos no tamanho do grafo. Isso está de acordo com (15) do Exercício 3, que diz:

$$T_{Prior}(n, m, r) = O(n^{r+2})$$

Como r faz parte do expoente, aumentos de r impactarão mais o tempo de processamento do que aumentos de n . De fato, para todo $n > 2$, o aumento de uma unidade no expoente será muito mais significativo do que o aumento de uma unidade na base. Como exemplo, podemos lembrar de que $10^2 = 100$ e $10^3 = 1000$, ao passo que $10^2 = 100$ e $11^2 = 121$.

Para verificarmos quantitativamente a conformidade entre a análise teórica e os resultados experimentais, tentaremos fazer uma associação entre os tempos de execução do algoritmo e as complexidades previstas teoricamente. Para isso, utilizaremos os resultados que obtivemos para a instância de exemplo do enunciado do problema, **ZikaZeroZ**, e para a instância **ZikaZeroZ13**.

Considerando que essas instâncias se aproximam, respectivamente, do melhor e do pior caso do algoritmo, podemos comparar a complexidade do tempo de execução (*determinada no Exercício 3*) ao tempo de execução observado (*apresentado no Exercício 5*), a fim de estimar as constantes envolvidas e, dessa forma, estimar teoricamente os tempos dos demais testes, comparando-os, posteriormente, com os tempos observados experimentalmente.

Assim, pelo Exercício 3, temos que:

$$T_{Melhor}(n, m, r) = O(nr + m) < c_1(nr + m) \quad (1)$$

$$T_{Prior}(n, m, r) = O(n^{r+2}) < c_2 n^{r+2} \quad (2)$$

Considerando (1) e os dados da instância **ZikaZeroZ** indicados no Exercício 5, temos:

$$10 < c_1(5 \cdot 4 + 6) \Rightarrow c_1 > \frac{10}{26} = \frac{5}{13}$$

$$\text{Faremos } c_1 = \frac{6}{13} \quad (3)$$

Considerando (2) e os dados da instância **ZikaZeroZ13** indicados no Exercício 5, temos:

$$3290 < c_2(5^{10+2}) \Rightarrow c_2 > \frac{3290}{244.140.625} = \frac{658}{48.828.125}$$

$$\text{Faremos } c_2 = \frac{659}{48.828.125} \quad (4)$$

Agora, calculando os tempos de cada uma das instâncias testadas e considerando os dados de cada uma delas (*ver Exercício 5*), podemos construir a seguinte tabela:

Instância	Resultados Experimentais						Previsões Teóricas	
	n	m	r	s	sPior	t	TMelhor	TPior
ZikaZeroZ	5	6	4	6	625	10	12	0
ZikaZeroZ10	5	0	10	9.765.625	9.765.625	1.810	23	3.295
ZikaZeroZ11	5	3	10	10.368	9.765.625	10	24	3.295
ZikaZeroZ12	5	3	10	4.608	9.765.625	10	24	3.295
ZikaZeroZ13	5	10	10	9.765.625	9.765.625	3.290	28	3.295
ZikaZeroZ20	50	50	5	1.080.450	312.500.000	490	138	10.544.000
ZikaZeroZ21	100	100	5	1.069.200	1,00E+10	730	277	1.349.632.000
ZikaZeroZ22	200	200	5	1.114.400	3,20E+11	1.320	554	1,73E+11
ZikaZeroZ23	1.000	1.000	5	1.080.000	1,00E+15	4.080	2.769	1,35E+16
ZikaZeroZ30	5	6	50	1.048.576	8,88E+34	890	118	3,00E+31
ZikaZeroZ31	5	6	100	1.048.576	7,89E+69	1.680	234	2,66E+66
ZikaZeroZ32	5	6	200	1.048.576	6,22E+139	3.250	464	2,10E+136
ZikaZeroZ33	5	6	1.000	1.048.576	-	15.910	2.310	-
ZikaZeroZ40	50	50	50	262.144	8,88E+84	2.830	1.177	3,00E+83
ZikaZeroZ41	100	100	100	262.144	1,00E+200	12.410	4.662	1,35E+199
ZikaZeroZ42	200	200	200	262.144	-	53.390	18.554	-
ZikaZeroZ43	1.000	1.000	1.000	262.144	-	1.733.780	462.000	-

Tabela 1 – Resultados experimentais e previsões teóricas relacionadas à complexidade temporal das instâncias testadas.

Na tabela 1, os campos que estão sem valor e que estão preenchidos com um traço, não puderam ser calculados em função do valor ser muito grande.

Nessa tabela, as colunas sob o rótulo *Resultados Experimentais* foram preenchidas com os dados do Exercício 5. Já os valores das colunas sob o rótulo *Previsões Teóricas*, foram calculados de acordo com (1), (2), (3) e (4), utilizando os dados de cada instância.

A coluna *s* indica o número de repetições feitas para cada instância. Conforme mencionamos no Exercício 3, *s* nos dá uma noção de quão esparsa é a matriz *R*, pois o número de repetições deriva do número de elementos com valor 1 nessa matriz. Assim, quanto mais esparsa for *R*, menor será o valor de *s* e menos repetições serão realizadas. Quanto menos esparsa for *R*, maior será o valor de *s* e mais repetições serão realizadas.

A coluna *sPior* apresenta o número de repetições que seriam realizadas para cada instância, caso a matriz *R* estivesse totalmente preenchida com valores 1. A coluna *sPior* está associada à coluna *TPior*. Como *TPior* é o limite superior para o tempo de processamento do algoritmo no pior caso, quanto mais próximo de *sPior* estiver o número de repetições *s* de uma instância, mais próximo o tempo de processamento *t* dessa instância estará do limite *TPior*.

Sendo assim, como dissemos no Exercício 3, no melhor caso do algoritmo, a matriz *R* será esparsa, ou seja, teremos um único voluntário por foco de reprodução. No pior caso, teremos uma matriz *R* totalmente densa, isto é, cada voluntário estará associado a todos os focos de reprodução.

A coluna *t* apresenta o tempo de execução do algoritmo, conforme a medição feita no Exercício 5. Para cada instância, ou seja, em cada linha da tabela, o valor de *t* deve estar abaixo dos valores das colunas *TMelhor* e *TPior*, ou entre eles. Estará mais próximo de *TMelhor*, caso a instância se

se aproxime mais do melhor caso. Estará mais próximo de $TPior$, caso ela esteja mais próxima do pior caso do algoritmo.

Pela Tabela 1, verificamos que o valor de t está mais próximo do valor $TMelhor$ do que do valor $TPior$ para quase todas as instâncias testadas. Isso pode ser visto mais facilmente no gráfico que se segue, no qual apresentamos uma comparação entre o tempo de processamento t e o tempo $TMelhor$ de cada instância.

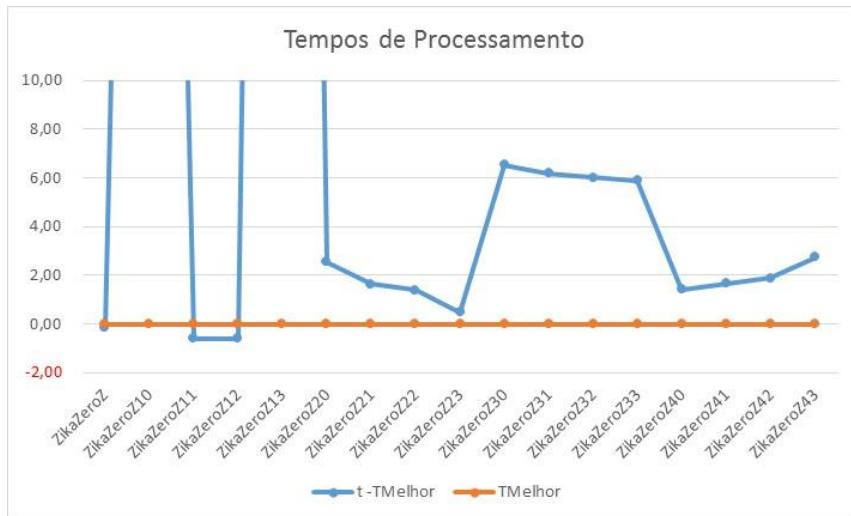


Figura 5 – Gráfico dos tempos de processamento das instâncias testadas.

Pelo gráfico, vemos facilmente que apenas os tempos observados para as instâncias 10 e 13 se distanciam consideravelmente de $TMelhor$. Realmente, criamos tais instâncias de acordo com o critério que define o pior caso do algoritmo (*matriz R totalmente densa*). Para essas instâncias, s é igual a $sPior$ e seus tempos de processamento estão mais próximos do limite superior $TPior$.

Já a instância de exemplo, *ZikaZeroZ*, e as instâncias 11 e 12 estão de acordo com o critério que define o melhor caso do algoritmo (*matriz R muito esparsa*). Por isso, pelo gráfico, vemos que os tempos de processamento dessas instâncias possuem $TMelhor$ como um limite superior.

As demais instâncias testadas representam casos intermediários de complexidade e estão entre o melhor e o pior caso do algoritmo. Todas elas, entretanto, estão mais próximas do melhor caso, pois a proximidade do pior caso, conforme mencionamos no Exercício 5, inviabiliza a utilização do algoritmo em instâncias com grandes volumes de dados. Apenas instâncias com poucos dados podem ser processadas em tempo viável estando próximas do pior caso.

Grandes instâncias, como a *ZikaZeroZ43*, quase sempre terão tempo de processamento grande, exceto se tiverem uma matriz R mínima (*matriz identidade por exemplo*). Contudo, esse tempo de processamento será razoavelmente viável somente se a sua matriz R for bem esparsa. É o que ocorre na instância *ZikaZeroZ43*. Pela Tabela 1, podemos ver que, apesar da instância ser grande, sua matriz R não é densa, pois s é substancialmente menor do que $sPior$, que, de tão grande, não foi calculado.

Por todas essas considerações a partir da Tabela 1, definida com base na análise teórica e nos resultados experimentais, acreditamos que, em relação à complexidade temporal do algoritmo, a análise teórica que fizemos e os resultados experimentais que obtivemos estão de acordo.

Com relação à complexidade espacial, indicamos no Exercício 3 que as principais estruturas de dados do algoritmo são R_{nxr} , G_{nxn} , S_{1xn} , $V_{0\ 1xr}$, V_{1xn} e $V_{i\ 1xr}$. Indicamos ainda que $Q(n,r)$, que é a função que nos dá a quantidade de memória utilizada pelo algoritmo, é dada por:

$$Q(n,r) = nr + n^2 + n + r + n + r = n^2 + nr + 2(n + r) \quad (5)$$

$$Q(n,r) = O(n^2 + nr) \quad (6)$$

Para verificarmos a conformidade entre a análise teórica e os resultados experimentais no que se refere à complexidade espacial, observamos a quantidade de memória utilizada pelo programa durante o processamento da instância *ZikaZeroZ43*. Apresentamos essa observação na Figura 6 que se segue, que é um recorte do Gerenciador de Tarefas do sistema operacional (*Windows*) da máquina na qual executamos o programa.

Nome	CPU	Memória	Disco	Rede
zikazeroz.exe	27,8%	8,5 MB	0 MB/s	0 Mbps

Figura 6 – Consumo de memória durante o processamento da instância *ZikaZeroZ43*.

Conforme apresentado na Figura 6, a quantidade de memória consumida pelo programa foi de 8,5 MBytes = 8.912.896 bytes.

Agora, considerando (5) e (6) e os dados da instância *ZikaZeroZ43* apresentados no Exercício 5, podemos construir a Tabela 2 que se segue:

Instância	Resultados Experimentais			Previsões Teóricas	
	n	m	r	$Q(n,r)$	$O(n^2+nr)$
ZikaZeroZ43	1.000	1.000	1.000	2.004.000	2.000.000 c_1

Tabela 2 – Resultados experimentais e previsões teóricas relacionadas à complexidade temporal da instância *ZikaZeroZ43*.

Pela Tabela 2, verificamos que $Q(n,r)$, que foi calculado por (5), indica a utilização de 2.004.000 elementos inteiros. Vemos ainda que $O(n^2 + nr)$ limita o consumo de memória em $2.000.000 \cdot c_1$ de elementos inteiros, onde c_1 é uma constante positiva.

Considerando que, na máquina em que realizamos os testes, um elemento inteiro ocupa 4 bytes na memória, temos que:

$$Q(n,r) = 8.016.000 \text{ bytes} \quad (7)$$

$$Q(n,r) = O(n^2 + nr) = 8.000.000c_1 \text{ bytes} \quad (8)$$

Como c_1 é uma constante positiva, vemos que (8) pode ser um limite superior para $Q(n,r)$ e ainda estar de acordo com o resultado experimental. Basta fazermos $c_1 > 1,114112$.

Sendo assim, acreditamos que a análise teórica que fizemos e os resultados experimentais que obtivemos também estão de acordo em relação à complexidade espacial do algoritmo.

Observações Finais:

1. Uma única consideração que fizemos em nossa análise teórica foi implementada de maneira diferente. Havíamos considerado que poderíamos verificar se um elemento pertence a um vetor, isto é, se um vértice pertence ao subgrafo induzido $G[V']$, em um tempo constante. Entretanto, na implementação, isso não ocorreu. A função *pertence*, que é a função que realiza essa tarefa no algoritmo implementado, possui complexidade $O(r)$ e não $O(1)$.

Isso altera a complexidade do algoritmo de busca que estamos utilizando, contudo, verificamos que tal alteração, que não foi considerada na teoria, não é suficientemente grande para mudar a complexidade do algoritmo como um todo, causando um conflito entre as previsões teóricas e as observações experimentais. Isso porque o algoritmo possui trechos de maior complexidade, que acabam se sobrepondo a essa diferença.

É possível alterar a função *pertence*, para que ela realize sua tarefa em $O(1)$. Contudo, em nosso entendimento, isso exigiria a criação de uma nova matriz de dimensão $1 \times n$, assim como o aumento da complexidade das funções que determinam os subconjuntos V' , para que essa nova matriz fosse preenchida a cada determinação de um subconjunto.

Assim, é possível que a complexidade do algoritmo como um todo não seja alterada com uma mudança na função *pertence*, ou até sofra uma alteração negativa qualitativamente. Entretanto, é certo que a complexidade espacial não seria alterada, permanecendo como indicamos em (16) no Exercício 3. Assim, deixaremos essa possibilidade de mudança para o futuro, como uma possível melhoria do algoritmo.

2. Outro aspecto importante a ser observado é a impossibilidade de utilizarmos o algoritmo em instâncias mais complexas do problema, devido ao tempo inviável de processamento.

Para resolvemos essa questão, precisaríamos alterar a forma de obter os subconjuntos V' . No algoritmo atual, obtemos esses subconjuntos a partir da matriz R , considerando cada uma das possíveis combinações de voluntários v capazes de cobrir todos os focos de reprodução f .

Em instâncias mais complexas, com voluntários acessando concomitantemente mais de um foco, várias dessas combinações poderiam ser ignoradas, ou aquelas que fossem mais óbvias poderiam ser consideradas e tratadas com antecedência.

Assim, acreditamos ser possível reduzir o tempo de processamento com a elaboração de heurísticas para a identificação e para o tratamento dos subconjuntos V' . Poderíamos também alterar totalmente a forma de obtenção desses subconjuntos, mas deixaremos isso como uma tarefa de otimização do algoritmo.

3. Da forma como está, o algoritmo garante a solução ótima do problema, embora ela não seja uma solução eficiente para todas as instâncias.

Relatório do Trabalho Prático de PAA: ZikaZeroZ

Daniel Kneipp de Sá Vieira

1 de maio de 2016

1 Exercício 1: Modelagem para o problema ZikaZeroZ

A modelagem proposta consiste uma lista de adjacência representando o grafo não direcionado de voluntários e relações de amizade $G(\mathbb{V}, \mathbb{A})$ e uma lista de listas D em que cada elemento especifica quais focos de dengue cada pessoa tem acesso. Caso $D(v_1) = \{f_2, f_3\}$, por exemplo, isso representa que o voluntário v_1 tem acesso aos focos f_2 e f_3 .

2 Exercício 2: Algoritmo

O algoritmo proposto para encontrar o menor grafo induzido conexo composto pelos voluntários que juntos acessam todos os focos é definido no Algorítimo 1.

Algoritmo 1: *obterV'*

Entrada: $G(\mathbb{V}, \mathbb{A})$, D
Saída: \mathbb{V}'

1 **início**

2 $D_{inv} \leftarrow inverterListaDeFocos(D);$

3 $P \leftarrow permutações(D_{inv});$

4 **para** cada $p \in P$ **faz**

5 $\left[\begin{array}{l} removerVoluntáriosRepetidos(p); \\ removerPermutaçõesRepetidas(P); \end{array} \right]$

6 $\mathbb{P}_c \leftarrow \emptyset;$

7 **para** cada $\mathbb{V}_f \in P$ **faz**

8 $\left[\begin{array}{l} G_p(\mathbb{V}_f, \mathbb{A}_f) \leftarrow grafoInduzido(G, \mathbb{V}_f); \\ \text{se } éConexo(G_p) \text{ então} \end{array} \right]$

9 $\left[\begin{array}{l} \mathbb{P}_c \leftarrow \{\mathbb{P}_c, \mathbb{V}_f\}; \\ \end{array} \right]$

10 $\left. \begin{array}{l} \\ \end{array} \right]$

11 $\left. \begin{array}{l} \\ \end{array} \right]$

12 $P_c \leftarrow ordenar(\mathbb{P}_c, soma(p \in \mathbb{P}_c));$

13 $\mathbb{V}' \leftarrow menor(P_c, tamanho(p \in P_c));$

14 **retorna** \mathbb{V}' ;

Recebendo como entrada o grafo de voluntários $G(\mathbb{V}, \mathbb{A})$ e lista de acessos aos focos D , o primeiro procedimento deste algoritmo (mostrado na linha 2) é construir uma nova lista de listas D_{inv} de focos por pessoas que tem acesso à estes focos (diferente de D em que se tem pessoas por focos que a pessoa tem acesso). Ou seja, $D_{inv} = \{f_1 \rightarrow \{v_1, v_2\}, f_2 \rightarrow \{v_1\}\}$ descreve que o foco f_1 pode ser acessado por v_1 e v_2 e o foco f_2 só pode ser acessado por v_1 . D_{inv} é construída para facilitar o procedimento feito na linha 2.

Posteriormente na linha 2 do Algorítimo 1 é obtido todas as permutações de voluntários que juntos acessam todos os focos usando o algoritmo 2 que faz o produto cartesiano de todas as listas de D_{inv} .

Algoritmo 2: *permutações*

Entrada: LL – Lista de listas com valores
Saída: PP – Permutações dos valores da listas

```

1  $PP \leftarrow \{\}$ ;
2 início
3   para cada lista  $L \in LL$  faça
4      $P_1 \leftarrow \emptyset$ 
5     para cada elemento  $e_{PP} \in PP$  faça
6        $P_2 \leftarrow \emptyset$ 
7       para cada elemento  $e_L \in L$  faça
8          $P_2 \leftarrow \{P_2, (\{e_{PP}, e_L\})\};$ 
9        $P_1 \leftarrow \{P_1, P_2\}$ 
10       $PP \leftarrow P_1$ 
11    retorna  $PP$ ;
```

Algumas permutações podem ter valores repetidos que precisam ser removidos e isso é feito na linha 5. Por conta da remoção de valores repetidos nas permutações algumas podem se tornar iguais. As permutações repetidas são removidas na linha 7.

Com isso P representa todas as combinações de voluntários que acessam todos os focos mas ainda deve-se verificar se estes se conhecem, ou seja, se o subgrafo induzido por eles em G é conexo. Para isso primeiramente na linha 10 se constrói o grafo induzido em G por cada permutação $\mathbb{V}_f \in P$ e depois verifica se este é conexo utilizando o algoritmo 3.

Algoritmo 3: éConexo

Entrada: $G(\mathbb{V}, \mathbb{A})$

Saída: $\begin{cases} \text{Verdadeiro,} & \text{se } G \text{ é conexo} \\ \text{Falso,} & \text{caso contrário} \end{cases}$

```
1 início
2    $v_s \leftarrow v \in \mathbb{V}_f;$ 
3    $\mathbb{V}_{BFS} \leftarrow BFS(G(\mathbb{V}, \mathbb{A}), v_s);$ 
4   se  $tamanho(\mathbb{V}_f) = tamanho(\mathbb{V}_{BFS})$  então
5      $\quad$  retorna Verdadeiro;
6   senão
7      $\quad$  retorna Falso;
```

O algoritmo 3 verifica se o grafo $G(\mathbb{V}, \mathbb{A})$ da seguinte forma: a partir de um vértice arbitrário $v \in \mathbb{V}_f$ é executado o algoritmo de busca em largura (*Breadth-First Search*) que retorna a lista de vértices \mathbb{V}_{BFS} que ele percorreu; caso \mathbb{V}_{BFS} tenha o mesmo número de vértices que \mathbb{V}_f , a busca em largura conseguiu percorrer todos os vértices, logo, o grafo é conexo.

Tendo encontrado as combinações de voluntários que acessam todos os focos de dengue e se conhecem, a linha 12 do Algoritmo 1 ordena cada combinação com base na soma dos índices dos valores que representam os voluntários. Ou seja, $ordenar(\{(2,5), (1,2)\}) \rightarrow ((1,2), (2,5))$.

Por fim a linha 13 obtém a menor permutação de voluntários, em que entende-se como menor permutação aquela que possui o menor número de voluntários. Caso haja empate, a primeira permutação encontrada das menores é selecionada. Por conta da ordenação feita na linha 12, caso haja empate, a menor permutação escolhida será aquela que possui a menor soma dos valores que representam seus voluntários.

3 Exercício 2: Análise de Complexidade

3.1 Análise Temporal

3.1.1 Análise do Algoritmo 2

Assumindo uma entrada contendo n conjuntos com k elementos cada (na realidade o número de elementos em cada conjunto pode variar), o Algoritmo 2 itera sobre cada conjunto de LL para gerar o produto cartesiano de maneira construtiva. A cada iteração do *loop* da linha 3 PP passa a ter o número de elementos que já tinha vezes o número de elementos do conjunto L que acabou de ser iterado, impactando no numero de execuções do *loop* da linha 5.

Ou seja, o *loop* mais interno executará $\sum_{i=1}^n k^i = \frac{k^{n+1} - 1}{k - 1} - 1$. O que implica que a complexidade do Algoritmo 2 para uma entrada com n conjuntos de tamanho k será $O(k^{n+1})$.

3.1.2 Análise do Algoritmo 3

O que o Algoritmo 3 faz é executar uma buscas em largura uma vez e fazer uma quantidade fixa de operações (comparação da cardinalidade de dois conjuntos), logo, o complexidade temporal deste algoritmo é dado pela própria busca em largura. Para um grafo de entrada com $|E|$ arestas e $|V|$ vértices, o Algoritmo 3 tem uma complexidade temporal de $O(|V| + |E|)$.

3.1.3 Análise do Algoritmo 1

Avaliando as operações mais custosas do Algoritmo 1 tem-se as linha 3, 9 e 10 que devem ser destacadas.

A operação na linha 3 possui um custo de $O(v^{|\mathbb{F}|+1})$, em que $|\mathbb{F}|$ é o número de focos e v é o número de voluntários que acessam os focos (assumindo que todos os focos são acessados pelo mesmo número v de pessoas).

Para cada permutação de voluntários obtida um subgrafo induzido de $G(\mathbb{V}, \mathbb{A})$ é gerado na linha 9 e é verificado se este é conexo na linha 10. Para uma única permutação, o custo de se gerar um subgrafo induzido é dado por $O(|\mathbb{V}_{sub}| + |\mathbb{A}_{sub}|)$, sendo que $|\mathbb{V}_{sub}| = v$ e $|\mathbb{A}_{sub}|$ são o número de vértices e arestas do subgrafo, respectivamente. Com isso o custo da verificação da conectividade do grafo é dado por $O(|\mathbb{V}_{sub}| + |\mathbb{A}_{sub}|)$ (assumindo um número de arestas constantes $|\mathbb{A}_{sub}|$ para todos os subgrafos). Já que estas duas operações são executadas $|P| = v^{|\mathbb{F}|}$ vezes, em que $|P|$ representa o número de permutações de P , o custo total gerado por essas duas operações é $O(|P|(|\mathbb{V}_{sub}| + |\mathbb{A}_{sub}|))$.

Portanto o custo total do Algoritmo 1 é $O(v^{|\mathbb{F}|+1} + v^{|\mathbb{F}|}(v + |\mathbb{A}_{sub}|))$. O pior caso deste algoritmo ocorre quando todos os voluntários conhecem todos os focos ($v = |\mathbb{V}|$) e todos os voluntários conhecem todos os outros ($|\mathbb{A}_{sub}| = |\mathbb{V}|^2$). Logo, o pior caso possui uma complexidade de $O(|\mathbb{V}|^{|\mathbb{F}|+1} + |\mathbb{V}|^{|\mathbb{F}|}(|\mathbb{V}| + |\mathbb{V}|^2)) = O(|\mathbb{V}|^{|\mathbb{F}|+1} + |\mathbb{V}|^{|\mathbb{F}|}|\mathbb{V}|^2)$.

Note que a complexidade gerada pela ordenação (linha 12) e outras operações não são explicitadas por serem desprezíveis se comparadas com a complexidade geral do algoritmo.

3.2 Análise Espacial

A operação que gera as permutações usadas no Algoritmo 1 (apresentada na linha 3) retorna $v^{|\mathbb{F}|}$ listas contendo $|\mathbb{F}|$ elementos cada, em que $|\mathbb{F}|$ é o número de focos e v é o número de voluntários que acessam cada foco (novamente, assumindo que cada foco é acessado pelo mesmo número v e voluntários). Com isso tem-se uma complexidade espacial gerada pela linha 3 igual a $O(|\mathbb{F}|v^{|\mathbb{F}|})$.

Para cada permutação gerada se reconstrói o subgrafo induzido em $G(\mathbb{V}, \mathbb{A})$ mas essa reconstrução não é feita para todas as permutações ao mesmo tempo, cada subgrafo é criado e destruído antes da criação do próximo. Logo, a complexidade espacial gerada pela linha 9 do Algoritmo 1 é dada por $O(|\mathbb{V}_{sub}| + |\mathbb{A}_{sub}|)$ (caso todos os subgrafos tenham o mesmo número de vértices e arestas), sendo $|\mathbb{V}_{sub}| = v$ e $|\mathbb{A}_{sub}|$ são o número de vértices e arestas do subgrafo, respectivamente.

Como dito anteriormente, o pior caso deste algoritmo ocorre quando $v = |\mathbb{V}|$ e $|\mathbb{A}_{sub}| = |\mathbb{V}|^2$, sendo assim, o pior caso da complexidade temporal do Algoritmo 1 é $O(|\mathbb{F}||\mathbb{V}|^{|\mathbb{F}|} + |\mathbb{V}| + |\mathbb{V}|^2) = O(|\mathbb{F}||\mathbb{V}|^{|\mathbb{F}|} + |\mathbb{V}|^2)$.

4 Exercício 5: Testes

Para se testar o algoritmo proposto foram geradas instâncias especificamente para o pior caso do algoritmo, ou seja, todos os voluntários se conhecem ($|\mathbb{A}| = \frac{|\mathbb{V}|(|\mathbb{V}|-1)}{2}$) e tem acesso a todos os focos. As instâncias foram geradas variando o número de voluntários n de 3 a 9 e para cada n tem-se um número de focos r variando de 3 a 8.

A Figura 1 mostra o tempo de execução para cada instância.

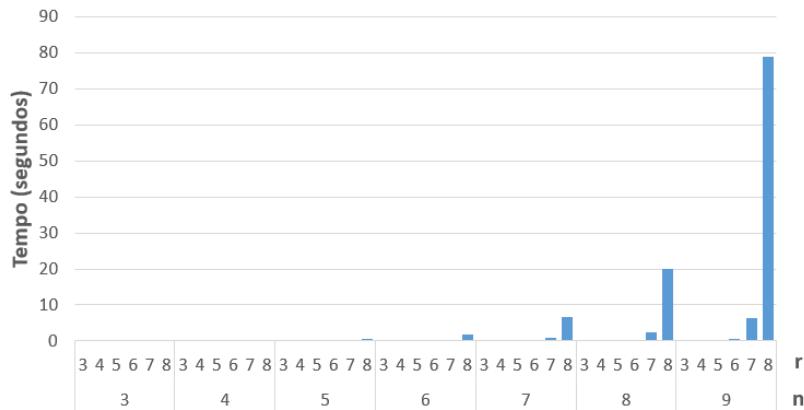


Figura 1: Comparação de tempos de execução para diferentes instâncias

5 Exercício 6: Análise dos Resultados

Observando a Figura 1 é possível notar o comportamento exponencial já previsto pela análise de complexidade realizada.

Um ponto interessante de se notar é a diferença no tempo de execução que o acréscimo no número de voluntários n faz em comparação com o acréscimo no número de focos r . A instância com $n = 8$, $r = 8$ teve um tempo de execução muito maior do que a instância com $n = 9$, $r = 7$, mostrando que o número de focos impacta muito mais o algoritmo do que o número de voluntários, o que entra em concordância com a análise de complexidade feita, já que $|\mathbb{F}|$ é o expoente.

Instâncias com um número de focos maior não puderam ser testados por conta do uso excessivo de memória.

Trabalho Prático de Projeto e Análise de Algoritmos

Tópicos em Grafos - ZicaZeroZ

Rodrigo Agostinho Chaves¹

¹ Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte – MG – Brasil

rodrigochaves@dcc.ufmg.br

1. Apresentação e Objetivos

Visando fortalecer o conhecimento de estruturas de dados e modelagem em grafos, foi proposto o desenvolvimento de um trabalho prático na disciplina de Projeto e Análise de Algoritmos. O trabalho contempla as mais diversas características de problemas da computação, o que requer uma boa modelagem dos dados, desenvolvimento de métodos para a solução, implementação, testes e análise. O objetivo principal é utilizar abordagens relacionadas as apresentadas em sala de aula.

O problema a ser resolvido é o **ZikaZeroZ**, que consiste em, a partir de um conjunto de usuários voluntários conectados em uma rede social, que tem acesso a um conjunto de focos de mosquito *Aedes aegypti*, propor um algoritmo para encontrar o número mínimo de pessoas que consigam alcançar todos os focos.

As seções seguintes abordarão as diversas etapas do processo de desenvolvimento e análise, que correspondem respectivamente a **Modelagem do Problema**, definição do **Algoritmo** utilizado, **Análise de Complexidade** temporal e espacial, **Descrição da Implementação** utilizada, aos **Testes Realizados** para validar o algoritmo e a discussão dos **Resultados Obtidos**.

2. Modelagem do Problema

A abordagem utilizada para modelar o problema **ZikaZeroZ** foi representar o conjunto de usuários da rede, suas respectivas conexões e os focos que cada um tem acesso como um grafo G . A modelagem de G é dada por $G(V, A)$ e uma relação $\mathcal{R}(v) : V \rightarrow F$ sendo:

- V o conjunto de todos os v vértices que representam usuários na rede social, ou $V = \{v_1, v_2, \dots, v_i\}$;
- A o conjunto de todas as arestas não direcionais e não ponderadas a existentes entre dois vértices distintos v e u pertencentes a V , ou $a = (u, v)$ e $A = \{a_1, a_2, \dots, a_j\}$. Cada aresta representa uma relação de amizade entre dois usuários distintos na rede social;
- F o conjunto de focos f de mosquito, ou $F = \{f_1, f_2, \dots, f_k\}$;
- $\mathcal{R}(v) : V \rightarrow F$ uma relação que representa os focos que cada usuário da rede tem acesso, ou $\mathcal{R}(v) = F_v$ sendo $F_v \subset F$ e $v \in V$.
- G cíclico (pode conter ciclos), não direcionado e isento de arestas repetidas.

Para representar a estrutura de dados do grafo G foi utilizado uma representação por uma matriz de adjacência, ou seja, uma matriz que represente a presença ou não de

uma aresta entre dois vértices. A matriz utilizada é quadrada, de dimensões tamanho iguais a $|V| \times |V|$. Cada elemento da matriz é definido pela equação 1, onde u e v são vértices de uma aresta (u, v) .

$$Adjacencia_{uv} = \begin{cases} 1 & \text{se } (u, v) \in A \\ 0 & \text{se } (u, v) \notin A \end{cases} \quad (1)$$

Para a representação da relação $\mathcal{R}(v)$ também foi utilizado um array bidimensional de dimensões $|V| \times |F|$. Cada elemento da matriz é definido pela equação 2, onde f é o numero do foco e u o vértice. Uma representação gráfica das estruturas de dados utilizados pode ser vista na figura 1.

$$Focos_{uf} = \begin{cases} 1 & \text{se a aresta } f \in \mathcal{R}(u) \\ 0 & \text{se a aresta } f \notin \mathcal{R}(u) \end{cases} \quad (2)$$

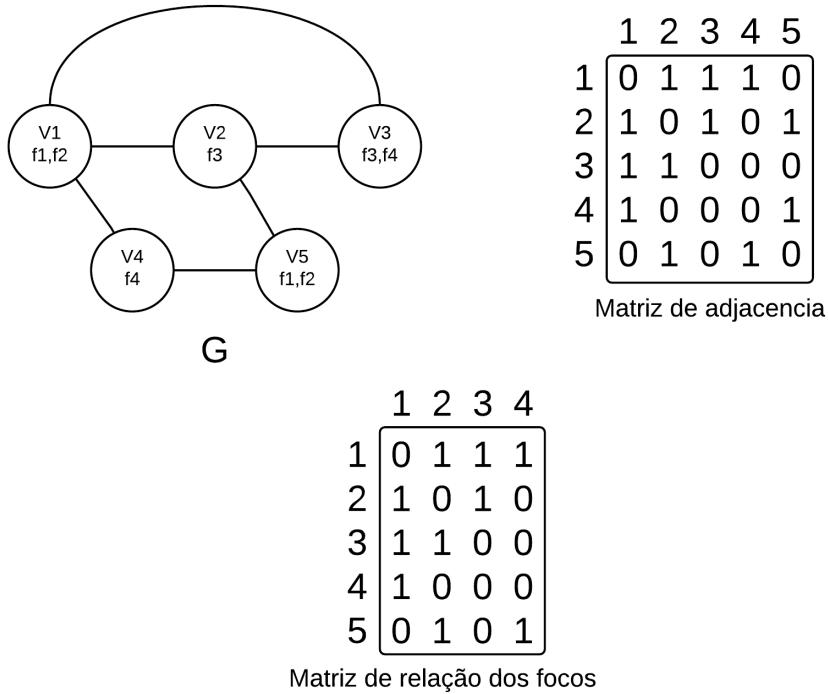


Figura 1. Exemplo da estrutura de dados utilizada para representar o grafo

A escolha de matrizes de adjacência para estruturas de dados levou em consideração dois fatores principais avaliados em [Cormen 2002] e [Gross and Yellen 2004]. O primeiro é a facilidade de estruturação dos dados utilizando tal representação, pois a verificação de existência de uma determinada aresta é direta ($O(1)$). O segundo fator é o baixo custo de se armazenar somente um byte por elemento da matriz, e o custo é invariável a quantidade de aresta. Estruturas que utilizam listas podem gastar mais memória caso o grafo seja denso, principalmente pela necessidade de se armazenar ponteiros.

A modelagem da solução para o problema consiste em uma busca sobre todos os subgrafos de G , para encontrar um G' que satisfaça as exigências do problema e seja o mínimo possível. Denominamos \mathbb{S} o conjunto de todos os subgrafos de G com um vértice ou mais, ou $\mathbb{S} = \{G'_1, G'_2, \dots, G'_r\}$ e $r = 2^{|G|} - 1$. A verificação da satisfação das restrições em cada grafo é polinomial e constituida de duas etapas: a verificação de conectibilidade de todos os vértices grafo e a cobertura de todos focos de mosquito.

Casos de empate, ou seja, caso exista dois ou mais elementos de \mathbb{S} que atendem as exigências do problema, serão utilizados os seguinte quesitos para avaliar o desempate entre dois subgrafos distintos G'_a, G'_b :

1. Será mantido G'_a se $|G'_a| < |G'_b|$. Ex: $V'_a = \{v_1\}, V'_b = \{v_1, v_2\}$;
2. Será mantido G'_a se a soma dos indices dos vertices de G presentes em G'_a for menor ou igual a de G'_b . Ex: $V'_a = \{v_1, v_2\}, V'_b = \{v_1, v_3\}$;
3. Será mantido G'_a se o vertice de menor índice pertencente a $(V'_a \cup V'_b) - (V'_a \cap V'_b)$ estiver presente em V'_a . Ex: $V'_a = \{v_1, v_4\}, V'_b = \{v_2, v_3\}$;
4. Será mantido G'_b se nenhuma das afirmativas anteriores forem satisfeitas.

3. O Algoritmo

Como mencionado anteriormente, o algoritmo proposto para a solução do **ZikaZeroZ** é uma busca exaustiva sobre todo o conjunto \mathbb{S} de subgrafos de G por um subgrafo G^* de custo mínimo que atende a todas as exigências. Dado que $|\mathbb{S}| = 2^{|G|} - 1$, temos que a busca realizada é de ordem exponencial.

Para cada $G' \in \mathbb{S}$ é verificado se o subgrafo atende a todos os requisitos para uma solução viável, e caso seja verdadeiro, é então verificado se G' tem custo melhor que a melhor solução já encontrada.

A verificação de viabilidade é dada por duas análises. Na primeira análise é avaliado se o subgrafo G' é conexo, através da busca em largura. O algoritmo 2 explica como e realizado tal procedimento. A segunda análise é realizada verifica se os vértices presentes em V' cobrem todos os focos do conjunto F , como mostrado pelo algoritmo 1. Sendo positivos para ambos os procedimentos, o grafo G' é determinado solução viável.

Algorithm 1: cobreTodosFocos

Data: $G'(V', A'), F$

Result: Booleano

```

1 begin
2   focosPresentes  $\leftarrow \emptyset$ 
3   foreach  $u \in V'$  do
4     focosPresentes  $\leftarrow$  focosPresentes  $\cup \mathcal{R}(u)$ 
5   if  $F \subset$  focosPresentes then
6     return Verdadeiro
7   place return False

```

Algorithm 2: isConexo

Data: $G'(V', A')$
Result: Booleano

```
1 begin
2   lista ← { $v'_1$ }
3   visitados ← ∅
4   while lista ≠ ∅ do
5     v ← pop(lista)
6     foreach u ← adjacente(v) do
7       if u ∉ lista and u ∉ visitados then
8         push(lista, u)
9       push(visitados, v)
10  if  $V' \subset visitados$  then
11    return Verdadeiro
12  return Falso
```

Após a verificação de viabilidade de G' , é então realizado a comparação com a melhor solução encontrada até o momento, e mantida a de melhor custo, de acordo com as funções de comparação definidas no fim da sessão 2. O algoritmo 3 representa a função principal utilizada para resolver o problema **ZikaZeroZ**. Uma melhoria realizada no código é o **if** presente na linha 5, onde verifica-se se o grafo a ser analizado já não cumpre uma das condições necessárias para se tornar uma solução, poupando então a computação da busca em largura e verificação de atendimento a todos os focos de mosquito.

Algorithm 3: Principal

Data: $G(V, A), F$
Result: V^*

```
1 begin
2   S ← geraTodosSubgrafos(G)
3   G* ← ∅
4   foreach  $G' \in S$  do
5     if  $G^* = \emptyset$  or  $|G'| \leq |G^*|$  then
6       if isConexo( $G'$ ) and cobreTodosFocos( $G', F$ ) then
7         G* ← melhorSolucao( $G^*, G'$ )
8   /*  $V^*$  é o conjunto de vértices de  $G^*$  */
```

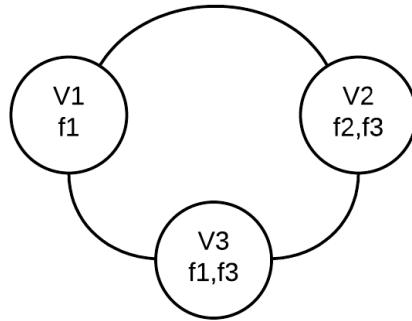


Figura 2. Exemplo de uma entrada G

Um exemplo de conjunto de vértices \mathbb{V} de todos os subgrafos em \mathbb{S} obtido a partir do grafo de entrada descrito na figura 2 pode ser visto na equação 3. As possíveis soluções presentes em \mathbb{S} são mostradas na equação 4. Por fim, temos a solução expressa na equação 5.

$$\mathbb{V} = \{\{v_1\}, \{v_2\}, \{v_3\}, \{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_1, v_2, v_3\}\} \quad (3)$$

$$validos(\mathbb{V}) = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_2, v_3\}\} \quad (4)$$

$$melhor(\mathbb{V}) = \{v_1, v_2\} \quad (5)$$

4. Análise de Complexidade

4.1. Análise de Complexidade Temporal

Considerando as estruturas de dados utilizadas, e as operações e procedimentos acima descritos, e considerando também a necessidade de adaptações do pseudo-código apresentado anteriormente durante a implementação em C++, podemos definir a complexidade do algoritmo como:

- **Criar um subgrafo de G :** Considerando que um subgrafo pode conter de 1 a $|V|$ vértices, definimos então a complexidade de criar um subgrafo como $O(|V|^2)$, visto que é utilizado uma matriz quadrada de ordem $|V| \times |V|$ para armazenamento dos dados, e a mesma precisa ser preenchida.
- **Verificar se G' é conexo:** Como G' pode conter até $|V|$ vértices e foi utilizado a busca em largura para verificar a conectibilidade de G' , temos a complexidade como $O(|V| + |E|)$. Sabemos também que estrutura de dados utilizada é uma matriz de adjacência, então $|E| = |V^2|$. Logo temos $O(|V| + |V|^2)$ ou simplesmente $O(|V|^2)$.
- **Verificar a cobertura de todos os focos:** Verificar se um determinado G' cobre todos os focos de mosquito requer a união de todas as listas de focos presentes

em cada vértice. Considerando que cada vértice tem uma lista com $|F|$ elementos $\in \{0, 1\}$, e a união de cada vértice é $O(|F|)$, então o custo de verificar a cobertura é $O(|V||F|)$.

- **Comparar duas soluções para encontrar a melhor:** As condições de comparação entre dois grafos G'_a e G'_b consideram basicamente os índices dos vértices, e utilizam, no pior caso, operações de união e disjunção entre conjuntos. O custo de se realizar tais operações é de $O(|V'_a| + |V'_b|)$. Logo, o custo maximo de uma operação de comparação entre dois grafos é $O(|V|)$.
- **Realizar a execução da função principal:** Tendo em vista que o conjunto \mathbb{S} contém $2^{|G|} - 1$ elementos, e para cada elemento G' desse conjunto, são realizadas as operações de verificação se G' é conexo, se cobre todos os focos e se é melhor que a solução até então encontrada, o que corresponde respectivamente as complexidades $O(|V|^2)$, $O(|V||F|)$ e $O(|V|)$. Considerando também que a construção de \mathbb{S} tem custo $O(2^{|G|}|V|^2)$, podemos calcular o custo total do algoritmo como:

$$\begin{aligned} & (2^{|G|} - 1) \times (O(|V|^2) + O(|V||F|) + O(|V|)) + O(2^{|G|}|V|^2) \\ & = O(2^{|G|}|V|^2 + 2^{|G|}|V||F|) \text{ ou } O(2^{|G|}(|V|^2 + |V||F|)) \end{aligned}$$

4.2. Análise de Complexidade Espacial

Para identificar o custo espacial é necessário analisar as estruturas de dados utilizadas para representar o grafo. Os custos de cada estrutura e geral do sistema são:

- **Custo de um grafo $G(V, A)$ qualquer:** Para representar o grafo foi utilizado uma matriz de adjacência de ordem $|V| \times |V|$ (figura 1). Por isso, podemos concluir que o custo de se armazenar um grafo é $O(|V|^2)$.
- **Custo de uma relação $\mathcal{R}(v) : V \rightarrow F$:** Para armazenar o conjunto de focos referente a cada vértice foi utilizado uma matriz de ordem $|V| \times |F|$, como descrito na figura 1. Logo, temos que a ordem de complexidade de armazenamento de uma relação é dada por $O(|V||F|)$.
- **Custo da estrutura G em conjunto com a relação \mathcal{R} :** Para facilitar uma estrutura com operações consolidadas, a estrutura de grafo implementada em C++ contém uma estrutura de representação da matriz de adjacência em conjunto com a matriz utilizada para representar a relação. Assim, temos a complexidade da estrutura completa para representar um grafo com todas as operações independentes de quaisquer outra estrutura como $O(|V|^2 + |V||F|)$.
- **Custo do conjunto \mathbb{S} com todos os subgrafos:** Considerando que o conjunto \mathbb{S} contém $2^{|V|} - 1$ subgrafos, representa-lo completamente geraria um custo de $O(2^{|V|}(|V|^2 + |V||F|))$, o que seria algo inviável de se utilizar para grandes entradas. Como a coexistência dos elementos de \mathbb{S} não é necessária, pois cada um é avaliado separadamente do outro, foi utilizado uma modelagem que cria o subgrafo somente durante sua utilização. Logo, é necessário um espaço para armazenar um G' , o que é $O(|V|^2)$. Também é necessário a utilização de um inteiro para armazenar a permutação do G' atual. Esse inteiro necessita de $O(|V|)$ bits, já que a permutação utilizada é um contador binário.
- **Custo de espaço total:** É utilizado então durante todo o programa quatro variáveis: G , G' , G^* e \mathbb{S} . Para G , G' e G^* são utilizados espaço da ordem de $O(|V|^2 + |V||F|)$, e um contador binário de $O(|V|)$ para representar \mathbb{S} . Temos que o custo de espaço total utilizado para o **ZikaZeroZ** é de $O(|V|^2 + |V||F|)$.

5. Descrição da Implementação

Para realizar a implementação do algoritmo proposto foi utilizado a linguagem C++ de forma Orientada a Objetos. Foram criadas duas classes, uma denominada *Graph*, para representar um grafo, e outra denominada *Contador*, que representa o conjunto \mathbb{S} , ou o contador binário para controlar as permutações dos subgrafo da entrada.

A classe *Graph* se localiza no arquivo *graph.h* e contém operações referentes a grafos e a estrutura necessária a solução, tais como construir um subgrafo a partir de um grafo e o subconjunto de vértices, inserir arestas, obter a quantidade de vértices, verificar se o mesmo é conexo e se abrange todos os focos de F .

A classe *Contador* contém um contador binário e operações de conversão do contador binário para um conjunto de vértices que possa ser interpretado pela classe *Graph*. A mesma foi implementada em um arquivo denominado *contador.h*.

O programa principal encontra-se no arquivo *main.cpp*, e contém a função main principal e uma outra função responsável pela obtenção dos dados de entrada.

Para compilar e executar o programa, é necessário estar utilizando um ambiente Linux e ter o compilador *g++* instalado. Os comandos necessários para executar através do terminal, e no diretório do programa são:

- **Compilar:** `$ compilar.sh`
- **Executar:** `$ executar.sh arquivo_de_entrada arquivo_de_saida`

6. Testes Realizados

Para validar os resultados, foram realizados dois tipos de testes, o teste de corretude do algoritmo, ou seja, verificar se o algoritmo alcança os resultados esperados, e o teste de desempenho para verificar se a complexidade calculada corresponde ao encontrado com a execução da solução implementada.

6.1. Testes de Corretude

Os testes de corretude avaliarão três grafos diferentes, e mostrará se o resultado obtido corresponde ao esperado. Alguns casos específicos serão mostrados, para averiguar se cada particularidade de uma solução é encontrada. Serão avaliados principalmente a identificação de todas as soluções viáveis e os quesitos de desempate utilizados para chegar a melhor solução. Para melhor visualização, o código foi alterado para não realizar a poda e fornecer todos as possíveis soluções.

O primeiro teste a ser apresentado tem como entrada o grafo representado na figura 3. Uma peculiaridade desse teste é a presença de duas soluções com a mesma quantidade de vértices, mas que na soma de seus índices proporciona o desempate. De todas os $2^6 - 1$ subgrafos possíveis, somente os seguintes foram determinados como soluções corretas, e o destacado como a melhor solução.

1. $\{v_1, v_2, v_3, v_4, v_6\}$
2. $\{v_1, v_2, v_3, v_5, v_6\}$
3. $\{v_1, v_2, v_3, v_6\}^*$
4. $\{v_2, v_3, v_4, v_6\}$
5. $\{v_2, v_3, v_4, v_5, v_6\}$

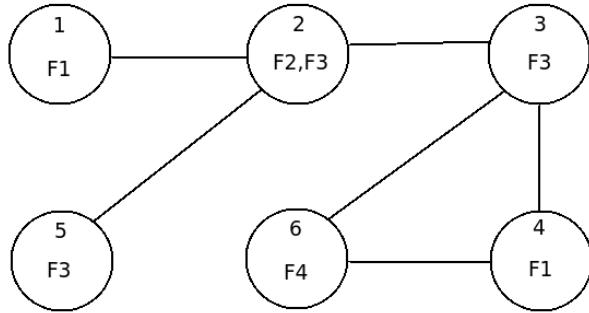


Figura 3. Teste de entrada com 6 vértices e 4 focos

Uma das peculiaridades do segundo exemplo é a existência de dois vértices que em conjunto acessam todos os focos de mosquito, mas não são conexos entre si. Para existir uma solução viável com ambos, é necessário então que haja um caminho entre ambos. Para tal, é necessário que um terceiro vértice seja acrescido na solução. O grafo com o Problema em questão pode ser visto na figura 4. Logo, a melhor solução para esse caso é dada pelos vértices $\{v_1, v_2, v_3\}$.

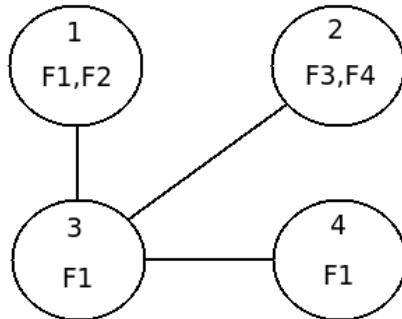


Figura 4. Teste de entrada com 4 vértices e 4 focos

Por fim temos a entrada representada pela figura 5. Nela há duas soluções com o mesmo número de vértices (v_1, v_4, v_2, v_3), e que a soma dos índices dos vértices também é igual (valor igual a 5). Nesse caso, o desempate é dado pela presença do vértice de menor índice e que não pertença a ambas as soluções. Nesse caso, a solução final considerada é v_1, v_4 .

É interessante ressaltar que os testes aqui expostos não provam que o algoritmo seja completamente correto, mas tem o objetivo de mostrar como o algoritmo se comporta em casos específicos. Testes mais completos foram realizados com entradas de diversos tamanhos, mas devido a dificuldade de representá-los de forma gráfica inviabiliza sua inserção nesse documento.

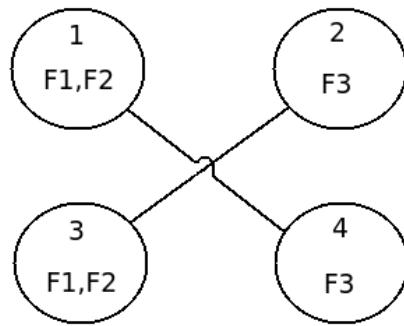


Figura 5. Teste de entrada com 4 vértices e 3 focos

6.2. Testes de Execução

A fim de verificar o comportamento de execução do algoritmo acima proposto implementado em uma linguagem de programação, foram realizados os testes de execução. Dois testes principais foram realizados, a fim de mostrar duas características específicas: a variação de comportamento em relação a idéia de verificar se a solução é melhor que a anterior antes de verificar se a mesma é viável; a variação em relação a quantidade de focos por vértices.

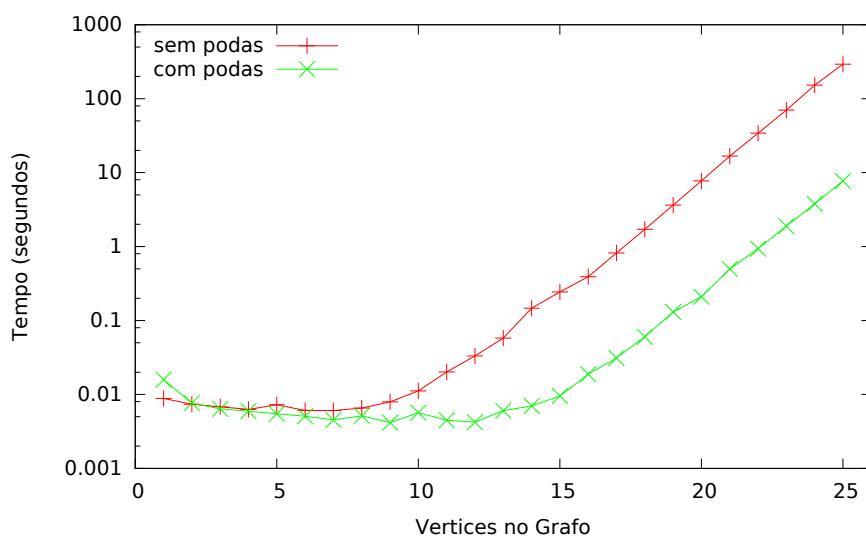


Figura 6. Comparação de tempo de execução realizando ou não podas em operações no subgrafo.

Para avaliar a melhoria da poda, ou seja, a não verificação de conectabilidade e abrangência de todos os focos em um subgrafo caso ele proponha uma solução pior que a já obtida, realizamos o seguinte teste: foi criado várias bases variando o tamanho do grafo em quantidade de vértices, e foram inseridas soluções ótimas que tenham baixa quantidade de vértices mais ao início da análise dos subgrafos, a fim de calcular logo no início boas soluções e realizarem várias podas. Para cada quantidade de vértices foram

realizados 10 execuções e calculado o custo médio das mesmas. Também foi realizado a execução das mesmas entradas removendo a condição que realiza a poda nas análises. A partir dos dados obtidos, foram gerados então o gráfico de tempo de execução representado pela figura 6.

Podemos observar que a execução com podas se comportou muito mais rápido que a sem podas, na ordem de 16 vezes, para entradas grandes. O grande fator que contribuiu para tal resultado foi a economia de operações ao descartar o cálculos desnecessários da busca em largura e verificação da cobertura de todos os focos. Embora os testes utilizaram entradas que possibilitavam a convergência rápida da solução, grafos que convergem lentamente tendem a ser consideravelmente mais rápidos, sendo quase sempre melhor que o pior caso.

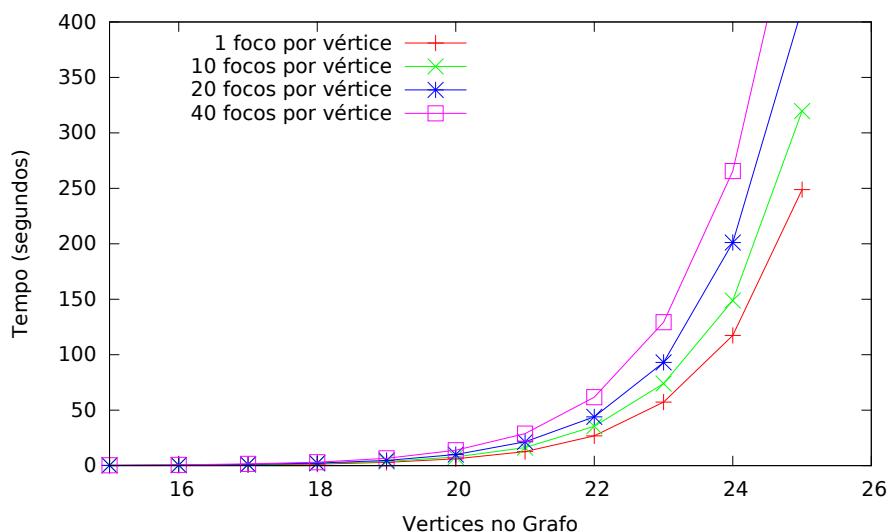


Figura 7. Comparação de tempo de execução variando a quantidade de vértices por foco.

O segundo teste levou em consideração a variação da quantidade de focos por vértice no grafo. Foram realizados testes utilizando 1, 10, 20 e 40 focos, e grafos densos. Para não haver influência da heurística da poda nos resultados, elas foram removidas do teste. A figura 7 representa os tempos de execução para tais algoritmos. É possível notar que quanto mais focos existem, mais caro é realizar o processamento dos mesmos nos grafos. Isso se decorre pois há um aumento na quantidade de dados armazenados e processados para verificar a cobertura em cada subgrafo.

7. Resultados Obtidos e Conclusões

Após a realização de vários testes, podemos observar que o algoritmo se comporta bem e é previsível em diversos cenários, mesmo tendo custo elevado para intâncias maiores. A busca pela solução entre todas as possibilidades pode ser extremamente custosa, e para esse tipo de problema, é a única que garante uma solução exata.

Analizando os tempos de execução para o pior e melhor caso, podemos verificar que a análise de complexidade se mantém se comparado ao obtido nos testes. Essa comparação pode ser vista na figura 8. foram definidas duas funções exponenciais

$f(|V|) = 2^{|V|}|V|^2$ e $g(|V|) = 2^{|V|}$, que limitam superiormente e inferiormente os tempos de execução obtidos, considerando a quantidade de focos não variante. As constantes utilizadas para verificar o comportamento assintótico das funções foram $c_1 = 2.0 \times 10^{-8}$ e $c_2 = 1.6 \times 10^{-7}$.

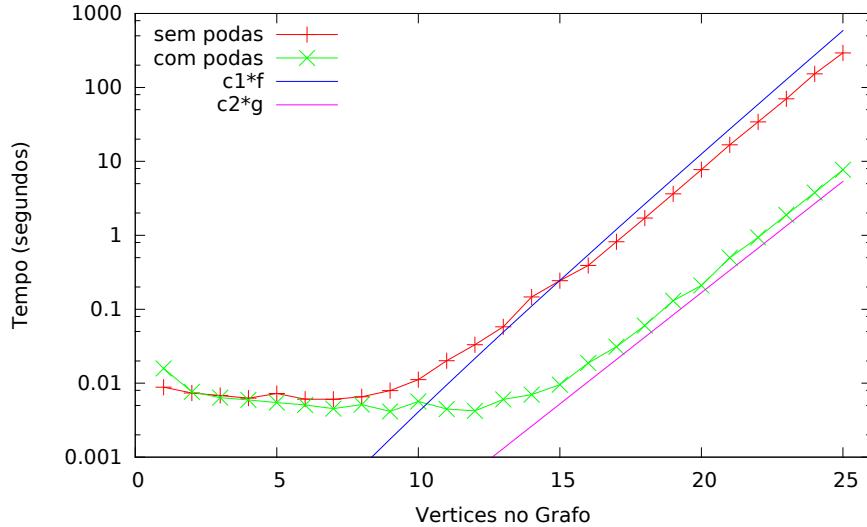


Figura 8. Comparação de tempo de execução em relação a equações exponenciais $f(|V|)$ e $g(|V|)$.

Considerando as análises assintóticas obtidas, podemos afirmar que o algoritmo se comporta assintoticamente comparável a $f(|V|)$ e $g(|V|)$. Assim, definimos pelos testes o tempo de execução limitados por $O(f(|V|))$ e $\Omega(g(|V|))$.

Os resultados obtidos permitiram perceber que a abordagem, apesar de gerar uma resposta exata, é inviável para solução de problemas reais, já que o tempo de execução para instâncias maiores são muito demorados. É interessante observar que uma alteração a estrutura para criar podas reduziu consideravelmente o custo de execução, mantendo o resultado ótimo. Por isso é possível considerar a utilização de heurísticas para a resolução do **ZikaZeroZ**, já que as mesmas podem gerar respostas com boa aproximação e em tempo polinomial.

Referências

- Cormen, T. H. (2002). *Algoritmos: teoria e prática*. Elsevier.
 Gross, J. L. and Yellen, J. (2004). *Handbook of graph theory*. CRC press.

Universidade Federal de Minas Gerais

Trabalho Prático - Grafos

Duílio Campos Sasdelli

Belo Horizonte
2016

[1. Introdução](#)

[2. Modelagem](#)

[2.1. Estruturas de Dados Utilizadas](#)

[2.2. Modelagem do Problema](#)

[3. Solução Proposta](#)

[3.1. Procedimentos Implementados](#)

[3.2. Análise Assintótica](#)

[3.2.1 - Tempo de Execução](#)

[3.2.2 - Utilização de Memória](#)

[3.3. Exemplo](#)

[4. Análise Empírica](#)

[4.1. Tempo de Execução](#)

[4.2. Memória](#)

[4.3. Corretude da Solução](#)

[5. Conclusão](#)

[6. Referências](#)

[Anexo I - Primeira Bateria de Testes](#)

[Anexo II - Segunda Bateria de Testes](#)

1. Introdução

O vírus da zika, causador da febre zika, é transmitido a humanos pelo mosquito *aedes aegypti*, mesmo vetor de doenças como a dengue e febre chikungunya. Considerada epidêmica, a febre zika pode ser combatida refreando-se a propagação do mosquito, o que se faz eliminando seus focos de reprodução.

O objetivo do presente trabalho é criar um algoritmo para auxiliar a escolha de uma rede colaborativa de voluntários responsável por combater os focos do mosquito *aedes aegypti*. O problema pode ser assim descrito¹:

Problema ZikaZeroZ

Dados um grafo $\mathbf{G(V;A)}$, em que \mathbf{V} é o conjunto de n voluntários e \mathbf{A} é o conjunto de m laços de amizade, um conjunto \mathbf{F} dos r focos de reprodução do mosquito, e, uma relação $\mathbf{R(v) : V} \rightarrow F$, definida para cada $v \in V$, explicitando os focos de reprodução aos quais o voluntário v tem acesso. O objetivo é selecionar o menor número de voluntários $V' \subseteq V$, tal que, todo foco é acessado, por pelo menos um voluntário $v \in V'$, e o grafo induzido por V' em \mathbf{G} é conexo.

2. Modelagem

A seção 2.1 explicará as estruturas de dados utilizadas e a seção 2.2 os problemas que deverão ser elucidados.

2.1. Estruturas de Dados Utilizadas

As estruturas de dados utilizadas para a resolução do problema foram modeladas utilizando-se dois grafos, quais sejam: um grafo G_v , representando os voluntários e seus laços de amizade; e um grafo bipartite G_{vf} , representando os voluntários e os focos os quais acessam. A divisão em dois grafos visou facilitar a implementação da solução do problema subdivindo-os em dois problemas menores, os quais serão discutidos em breve.

¹

Modelagem das Estruturas de Dados

$G_v (V; A)$: um grafo não direcionado com um conjunto V contendo $|V|$ vértices, correspondentes aos $|V|$ voluntários e um conjunto A contendo $|A|$ arestas na forma (u,v) em que $u, v \in V$. Se determinado par de voluntários u e v possuir um laço de amizade, então a aresta (u,v) apresentará um atributo peso $(u,v).p = 1$, caso contrário, $(u,v).p = 0$.

$G_{vf} (V_{vf}; A_{vf})$: um grafo não direcionado bipartite com um conjunto $V_{vf} = V \cup F$ contendo $|V + F|$ vértices, correspondentes aos $|V|$ voluntários e $|F|$ focos, e um conjunto A_{vf} contendo $|A_{vf}|$ arestas na forma (v,f) em que $v \in V$ e $f \in F$. Se determinado voluntário v acessar um foco f , então a aresta (v,f) apresentará um atributo peso $(v,f).p = 1$, caso contrário, $(v,f).p = 0$.

Nota-se que as arestas do grafo G_v não possuem peso e o grafo é não-direcionado, ou seja, para a sua implementação foi utilizada uma matriz de adjacência $|V| \times |V|$ na qual utilizou-se apenas os elementos acima da diagonal da matriz.

As arestas do grafo G_{vf} também não possuem peso e o mesmo é não direcionado e bipartite, ou seja, para a sua implementação, utilizou-se uma matriz de adjacência $|V| \times |F|$, economizando, consequentemente, memória. Assim, cada entrada da matriz de adjacência M apresentará a forma $T_{vf} = (v,f).p$.

O exemplo 1 mostrado a seguir possui 5 voluntários e 4 focos. São mostradas a definição formal, a representação gráfica e as matrizes de adjacência.

Exemplo 1:

$G_v (V; A)$:

$$V = \{v1, v2, v3, v4, v5\}$$

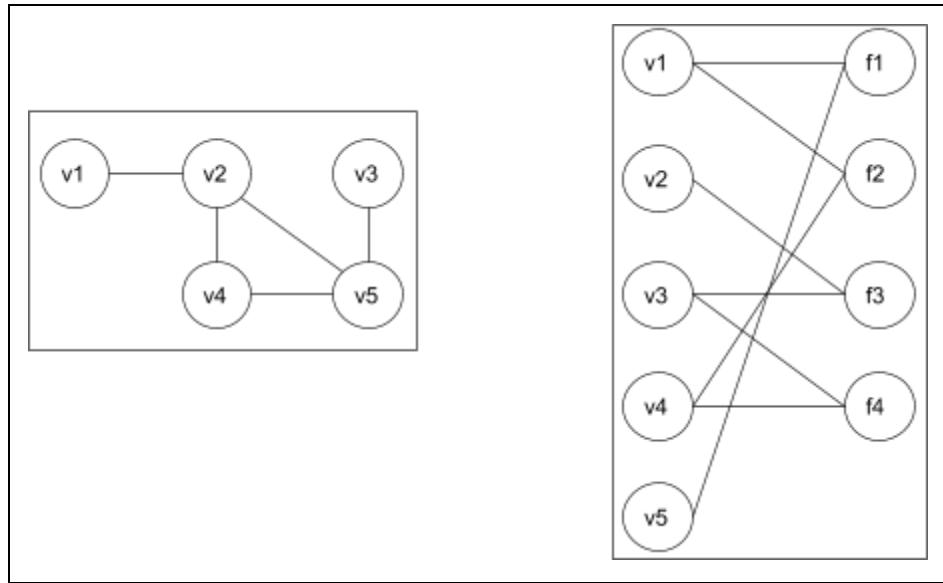
$$A = \{(v1,v2), (v2,v4), (v2,v5), (v3,v5), (v4,v5)\}$$

$G_{vf} (V_{vf}; A_{vf})$:

$$V_{vf} = \{ V = \{v1, v2, v3, v4, v5\} \cup F = \{f1, f2, f3, f4, f5\} \}$$

$$A_{vf} = \{(v1,f1), (v1,f2), (v2,f3), (v3,f3), (v3,f4), (v4,f1), (v4,f2), (v5,f1)\}$$

Exemplo 1 - Representação Gráfica de G_v e G_{vf} :



Exemplo 1 - Matrizes de Adjacência de G_v e G_{vf} :

0 1 0 0 0	1 1 0 0
0 0 0 1 1	0 0 1 0
0 0 0 0 1	0 0 1 1
0 0 0 0 1	0 1 0 1
0 0 0 0 0	1 0 0 0

2.2. Modelagem do Problema

A divisão das estruturas de dados em dois grafos foi proposital e se deve aos dois problemas distintos que devem ser resolvidos. No grafo G_{vf} ($V_{vf}; A_{vf}$) o problema a ser sanado é o de **hitting set mínimo**² ou **cobertura de conjuntos**. Já no grafo G_v ($V; A$) o problema a ser sanado é o de averiguar a **conectividade** de um dado subconjunto de vértices gerado pela solução do problema no grafo G_{vf} . A seguir será definido de forma mais precisa cada um desses problemas.

² Explicação disponível em [3].

O problema de **hitting set mínimo** pode ser definido, informalmente, como o de se encontrar uma coleção (S_1, S_2, \dots, S_n) de tamanho mínima de subconjuntos de um dado conjunto S , tal que a união dos elementos dessa coleção seja igual a S . Modelado por meio de um grafo bipartite, o objetivo é encontrar o número mínimo de vértices de um lado do grafo tal que todos os outros vértices do outro lado sejam cobertos. No caso em tela, o objetivo é formalmente descrito a seguir.

Hitting Set Mínimo

Dado um grafo bipartite $G_{vf} (V_{vf}; A_{vf})$ conforme definido na seção anterior, o problema é de se encontrar um subconjunto de tamanho mínimo V' de V (lembrando que $V_{vf} = V \cup F$ e $V \cap F = \emptyset$) tal que o conjunto de focos induzido F' , cujos elementos f sejam parte de alguma aresta (v,f) de A_{vf} para algum v de V' , seja igual a F .

O objetivo do problema é o de se encontrar um subconjunto de tamanho mínimo de vértices voluntários de tal modo que haja uma aresta desse conjunto para todos os vértices focos. Trata-se de um problema NP-completo³ pertencente à lista de 21 problemas de Karp com complexidade $O(2^n)^4$.

Logo após encontrado um subconjunto de vértices voluntários, deve-se atestar se esse subconjunto é conexo, ou seja, se há um conjunto de laços de amizade conectando-os. Mais formalmente, pode-se descrever o problema por meio do problema de se atestar se o grafo induzido gerado pelos subconjuntos de vértices é conexo.

Grafo Conexo

Dado um grafo $G (V; A)$, para todo vértice v de V deve haver um caminho p formado por um conjunto A' de A para todos os demais vértices $V - \{v\}$.

O problema de se atestar a conectividade pode ser resolvido facilmente utilizando-se uma busca em profundidade em tempo linear $\Theta(|V| + |A|)$.⁵

³ Ver [2] pp. 1117.

⁴ Cada voluntário V pode estar ou não presente em determinada solução, ou seja, é fácil perceber que cada voluntário pode assumir dois estados. Como há n voluntários, há 2^n combinações diferentes de configurações para se testar

⁵ Ver [2] pp. 606.

3. Solução Proposta

A implementação da solução fez uso da linguagem JAVA (SDK 1.8.0.77) em ambiente Windows por meio de uma IDE Eclipse Luna Service Release 1a (4.4.1). A linguagem JAVA foi utilizada porque ser mais fácil e rápida de implementar.

Para a resolução do problema, optou-se por utilizar uma função recursiva que incrementa a lista de voluntários escolhidos, testando, a cada nível de recursão, se essa lista de voluntários é capaz de cobrir todos os focos. Além disso, a cada nível de recursão, é construída uma fila de prioridades de voluntários ordenada de acordo com o número de focos novos cobertos. Trata-se de uma heurística gulosa: são escolhidos primeiros os voluntários que mais focos novos cobrem. Ademais, a cada nível de recursão, são utilizadas duas heurísticas:

Heurística 1:

Caso o número de voluntários escolhidos ultrapasse o valor da melhor solução, isso é, a com menor número de voluntários, a recursão não mais ultrapassará esse nível de recursão.

Heurística 2:

O voluntário corrente e os demais na árvore recursiva são descartados caso o voluntário escolhido não seja conexo com os voluntários escolhidos anteriormente.

Na seção seguinte o procedimento será explicado com mais detalhes.

3.1. Procedimentos Implementados

As variáveis globais (Pseudocódigo 1) utilizadas são definidas no quadro que se segue. Ressalta-se que as variáveis não são exatamente globais, visto que foram implementadas como atributos de uma classe criada para a resolução do problema. Os grafos **grafoV** e **grafoVF** correspondem aos grafos **G_v** e **G_{vf}** definidos na seção anterior. O conjunto

sVoluntarios armazena as soluções presentemente escolhidas. Já a variável **tMelhor** armazena a melhor solução presentemente escolhida.

Pseudocódigo 1 - Variáveis Globais:

```
1 Grafo grafoV; /* Grafo de Voluntários */
2 Grafo grafoVF; /* Grafo de Voluntários x Focos */
3 Int nV = grafoV.nV; /* Número de Voluntários */
4 Int nF = grafoVF.nF; /* Número de Focos */
5 Conjunto<Conjunto<Int>> sVoluntarios = []; /* Conjunto de Soluções */
6 Int tMelhor = INFINITO; /* Tamanho da Melhor Solução */
```

O procedimento de solução (Pseudocódigo 2) do problema começa inicializando cada posição dos vetores **fCobertos** e **vEscolhidos** como falsa. Esse vetores representam os focos cobertos e os voluntários escolhidos em um dado momento. Em seguida, é feita uma chamada do procedimento recursivo **cobreFocos**.

Pseudocódigo 2 - Procedimento ZikaZeroZ:

```
1 procedimento ZikaZeroZ()
2 {
3     Boolean[] fCobertos ← [ ];
4     Boolean[] vEscolhidos ← [ ];
5     para(i de 1 : nV)
6         fCobertos[i] ← false;
7     para(i de 1 : nF)
8         vEscolhidos[i] ← false;
9     cobreFocos(fCobertos,0,vEscolhidos,0);
10 }
```

O procedimento recursivo **CobreFocos** (Pseudocódigo 4) inicia testando se já atingiu-se um nível de recursão em que não é mais possível encontrar um conjunto de voluntários menor ou igual ao corrente (linhas 3 e 4). Trata-se da primeira heurística discutida anteriormente nessa mesma seção. Em seguida (linhas 5 e 6) é criada uma fila de prioridades em que se armazena o índice dos voluntários ainda não escolhidos e o número de focos ainda descobertos que esses voluntários cobrem. O procedimento de criação da fila de prioridades é mostrado no Pseudocódigo 3.

Pseudocódigo 3 - Procedimento CriaFilaDePrioridades:

```
1 procedimento criaFilaDePrioridades(filaV, fCobertos, vEscolhidos)
2 {
```

```

3     para i de 1 : grafoVF.nV
4     {
5         int sFocos ← 0;
6         se(vEscolhidos[i] = false)
7         {
8             para j de 1 : nF
9                 se fCobertos[j] = false
10                sFocos ← sFocos + grafoVF[i,j];
11            }
12            filaV.adiciona(Par(j,sFocos));
13        }
14    }

```

O procedimento **CobreFocos** então itera (linhas 7 a 34) sobre todos os voluntários presentes na fila de prioridades criada (**filaV**). A cada iteração, remove-se o primeiro voluntário da fila, isso é, o que cobre o maior número de focos descobertos (linha 9). São criadas cópias dos vetores **fCobertos** e **vEscolhidos** de modo que não haja conflito em iterações posteriores e não seja necessário resetar seus valores (linhas 10 e 11).

Em seguida, é realizada a segunda heurística (linhas 12 e 13), na qual testa-se a conectividade do vértice atual com os demais vértices escolhidos (Pseudocódigo 5). Essa heurística garante o invariante de que todo conjunto de voluntários escolhidos será necessariamente conexo a qualquer momento do algoritmo. Assim, não é mais necessário testar a conectividade após encontrados os conjuntos mínimos de voluntários que cobrem todos os focos.

Pseudocódigo 4 - Procedimento CobreFocos:

```

1 procedimento cobreFocos(fCobertos, nFC, vEscolhidos, nVE)
2 {
3     se(tMelhor < nVE + 1)
4         retorna;
5     FilaDePrioridades<Par<int,int>> filaV ← [ ];
6     criaFilaDePrioridades(filaV, fCobertos, vEscolhidos);
7     enquanto(!vazia(filaV))
8     {
9         Par<int,int> melhorV ← removePrimeiro(filaV);
10        Boolean[] fCobertosN ← fCobertos;
11        Boolean[] vEscolhidosN ← vEscolhidos;
12        se(nVE > 0 & !testaC(vEscolhidosN , melhorV.j))
13            continue;
14        para(i de 1 :nF)
15        {
16            se(grafoVF[melhorV.j,i] = 1)
17                fCobertosNovo[i] ← true;
18        }
19        vEscolhidosN[melhorV.j] ← true;

```

```

20     se(nFC + melhorV.sFocos = nF)
21     {
22         Set<Integer> sv ← [ ];
23         para(i de 1 :nV)
24         {
25             se(vEscolhidosN[i] = true)
26                 sVoluntario.adiciona(i);
27             }
28             sVoluntarios.adiciona(sv);
29             tMelhor = sv.tamanho();
30             continue;
31         }
32         senão
33             coverFocos(fCobertosN,nFC+vMax.sFocos,vEscolhidosN,nVE+1);
34         }
35     }

```

Passadas essas etapas, o voluntário pode, finalmente, ser escolhido e os focos presentemente cobertos podem ser atualizados (linhas 14 a 19). Caso a adição do voluntário escolhido realize a cobertura de todos os focos (linha 20), então trata-se de uma solução, uma vez que ela é, no mínimo, igual à melhor solução corrente, conforme testado anteriormente (linhas 3 e 4). A solução encontrada é então construída (linhas 22 a 27), adicionada ao conjunto de soluções (linha 28), o valor de **tMelhor** é atualizado (linha 29). Em seguida, o algoritmo prossegue para o próximo voluntário nesta mesma recursão, o qual também pode ser uma solução (nesse caso, o algoritmo considera o estado da recursão em seu início por meio dos vetores **vEscolhidosN** e **fCobertos**, desconsiderando-se o voluntário recém adicionado). Finalmente, caso todos focos ainda não tenham sido cobertos, é realizada uma nova chamada de **CobreFocos** já com o novo voluntário adicionado.

Pseudocódigo 5 - Procedimento TestaC:

```

1 procedimento testaC(vEscolhidos,v)
2 {
3     para(i de 1 :nV)
4         se(vEscolhidosN[i] = true & i <> v & grafoV[i,v] = 1)
5             retorna true;
6         retorna false;
7     }

```

3.2. Análise Assintótica

Apesar das heurísticas utilizadas, trata-se de um problema NP-Completo de ordem de complexidade de execução assintótica exponencial. Essa seção iniciará calculando a ordem de

complexidade de execução para os procedimentos implementados. Em seguida, será calculada a ordem de complexidade de memória para as estruturas utilizadas e as chamadas recursivas armazenadas na pilha (*stack*).

3.2.1 - Tempo de Execução

O procedimento **TestaC** possui complexidade assintótica $O(|V|)$ pois, no pior caso, necessita realizar até $|V|$ operações $O(1)$ (linhas 3 e 4), a fim de verificar se o voluntário v é adjacente aos voluntários já escolhidos (**vEscolhidos**). Nota-se que o procedimento de se verificar se uma aresta existe ($grafo[i,v] = 1$) possui complexidade $O(1)$ para matrizes de adjacência.

O procedimento **CriaFilaDePrioridades** é responsável por criar um Heap, estrutura que possui o procedimento de inserção com complexidade $O(\log(n))$ ⁶ (linha 12). Como são inseridas $|V|$ voluntários, a complexidade é $O(|V|\log(|V|))$. No entanto, o procedimento também necessita verificar, para todos os voluntários, a situação de cada foco (coberto ou descoberto), ou seja, a complexidade desse procedimento torna-se $O(|V|\log(|V|) + |V||F|)$ (linhas 6 a 10).

Finalmente, o procedimento **CobreFocos** é um procedimento recursivo que, na pior das hipóteses (linha 33), vai apresentar um nível de recursão igual ao número de voluntários. Isso se deve ao fato de que, a cada nível n de recursão, é testada uma solução n voluntários já escolhidos e ainda há $|V| - n$ por escolher. Assim, a cada nível de recursão n há, no máximo, outras $|V| - n$ chamadas recursivas, ou seja, o total de chamadas é menor quanto o mais profunda a recursão se encontra. Então, o número de chamadas de **CobreFocos**, no pior caso, será dado por:

$$\prod_{i=1}^{|V|} i = (|V|)(|V|-1)\dots(1) = |V|! .$$

A cada chamada recursiva, o algoritmo realiza sempre uma chamada de **CriaFilaDePrioridades**, a qual possui complexidade $O(|V|\log(|V|) + |V||F|)$, além dos seguintes procedimentos dentro do laço que itera sobre os elementos da fila (linha 7):

- **removePrimeiro** (linha 9): complexidade $O(1)$;
- **testaC** (linha 12): complexidade $O(|V|)$;
- cópias de vetores (linhas 10 e 11): complexidades $O(|F|)$ e $O(|V|)$;

⁶ <https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html>

- atribuições de vetor (linhas 14 a 18): complexidade $O(|F|)$;

Ressalta-se que, no pior caso, a condição que verifica que a solução fora encontrada (linha 20) nunca será executada, pois nunca haverá uma solução pois o algoritmo sempre realiza uma nova chamada recursiva. Assim, sem perda de precisão, a complexidade dos procedimentos realizados dentro dessa condição serão ignorados. Finalmente, vale ressaltar que a laço (linha 7) que itera sobre a fila é executado n vezes, sendo n o nível da recursão. Também sem perda de precisão, considerará-se que esse laço será executado $|V|$ vezes. Assim, chega-se à seguinte complexidade para o procedimento **CobreFocos**:

$$\begin{aligned}
 |V|! (O(|V|\log(|V|)) + |V||F|) + |V|(2 O(|F|) + O(|V|)) &= \\
 |V|! (O(|V|\log(|V|)) + |V||F|) + (O(|V||F|) + O(|V|^2)) &= \\
 |V|! (O(|V||F|) + O(|V|^2)) &= \\
 O(|V|! (|V||F| + |V|^2)) &= \\
 O(|V|!)
 \end{aligned}$$

Para um grande número de voluntários, não entanto, a complexidade aproxima-se de $O(|V|!)$. Ressalta-se que a complexidade do problema de **hitting set** é dada por $O(2^n)$ conforme já explicado anteriormente. Sabe-se que o procedimento implementado realiza buscas desnecessárias testando mais de uma vez uma mesma solução. No entanto, decidiu-se por assim fazê-lo em virtude do modo o qual a solução foi projetada, realizando recursões sobre e escolhendo voluntários ainda não inseridos na solução corrente. Conforme ver-se-á na seção seguinte, as heurísticas implementadas reduzirão consideravelmente o tempo de execução no caso médio.

3.2.2 - Utilização de Memória

Em relação à utilização de memória, será considerado o número máximo de chamadas recursivas de **CobreFocos** empilhadas, ou seja, $O(|V|)$.

Cada chamada recursiva de **CobreFocos** realiza a alocação de seus argumentos **nFC** e **nVE** (ambos inteiros de 4 bytes, $O(1)$) e ponteiros para os vetores **fCobertos** e **vEscolhidos** (ambos endereços de 8 bytes, $O(1)$). Além disso, também realiza a alocação de uma fila de

prioridades (linha 5, par de inteiros de 4 bytes, $O(|V|)$), dois vetores (**fCobertosN** e **vEscolhidosN**) utilizados nas recursões subsequentes (linhas 9 e 10, $O(|F|)$ e $O(|V|)$, respectivamente), um par de inteiros (linha 9, $O(1)$). Assim, cada chamada de **CobreFocos** utiliza $O(|F| + |V|)$ de memória e pode ser chamada até $|V|$ vezes, uma vez para cada voluntário selecionado. Assim, tem-se que o gasto de memória máximo desse procedimento é $O(|F||V| + |V|^2)$.

Em relação às variáveis globais, destaca-se que são utilizados dois grafos implementados com matrizes de adjacência, **grafoVF** que utiliza $|V||F|$ bytes de memória e o **grafoV**, que utiliza $|V|(|V|-1)/2$ bytes de memória, três inteiros (linhas 3,4 e 6) e um conjunto de conjunto de soluções, que poderá conter até $2^{|V|}$ elementos (combinações de voluntários), cada um armazenando até $|V|$ bytes, ou seja, $2^{|V|}|V|$. Na prática, contudo, o conjunto não crescerá, uma vez que as melhores soluções serão encontradas primeiro e as demais serão descartadas. Inobstante a isso, a complexidade de memória deve considerar o pior caso. Finalmente, o procedimento **ZikaZeroZ** cria dois vetores com tamanhos $|V|$ e $|F|$, ou seja, possui complexidade $O(|F| + |V|)$.

O total gasto de memória é, portanto, a soma dos procedimentos **ZikaZeroZ** e **CobreFocos** com as variáveis globais alocadas:

$$O(|F| + |V|) + O(|F||V| + |V|^2) + O(2^{|V|}|V| + |F||V| + |V|^2) = O(2^{|V|}|V|)$$

Contudo, é de se ressaltar que o pior caso exponencial de uso de memória raramente ocorre, ou seja, na maioria das vezes a complexidade recai em $O(|V|^2 + |F||V|)$, sendo, portanto, polinomial.

3.3. Exemplo

Como exemplo da execução do algoritmo, será utilizado o Exemplo 1, dado por:

Exemplo 1:

G_v (V; A):

$V = \{v1, v2, v3, v4, v5\}$

$A = \{(v1,v2),(v2,v4),(v2,v5),(v3,v5),(v4,v5)\}$

G_{vf} (V_{vf}; A_{vf}):

$V_{vf} = \{ V = \{v1,v2,v3,v4,v5\} \cup F = \{f1,f2,f3,f4,f5\}\}$

$$A_{vf} = \{(v1,f1), (v1,f2), (v2,f3), (v3,f3), (v3,f4), (v4,f1), (v4,f2), (v5,f1)\}$$

Segue a explicação da execução passo a passo (de forma resumida) do algoritmo considerando apenas duas primeiras recursões até a primeira solução ser encontrada:

- A primeira chamada de **CobreFocos** cria a lista de prioridades $\{(v1,2), (v3,2), (v4,2), (v3,1), (v5,1)\}$. Extrai-se, portanto $v1$ da lista e prossegue-se com uma nova chamada recursiva de **CobreFocos**;
- A segunda chamada de **CobreFocos** cria a lista $\{(v3,2), (v3,1), (v4,0), (v5,0)\}$. Extrai-se $v3$ e encontra-se uma solução, pois os quatro focos são cobertos com $v1$ e $v2$.
- O algoritmo então prossegue procurando novas soluções até o segundo nível de recursão (pois esse é o tamanho da melhor solução).

4. Análise Empírica

Para a análise empírica, foram realizados testes utilizando como métricas tanto o tempo de execução quanto a quantidade de memória utilizada. Foi utilizado um microcomputador com as seguintes características:

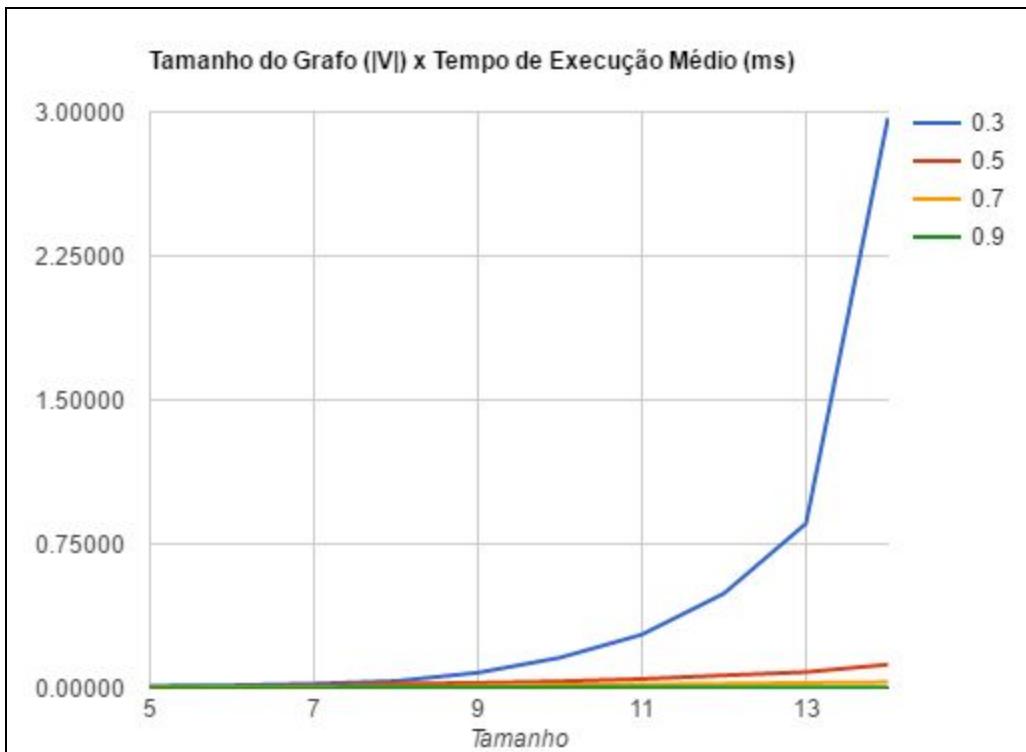
- Processador Intel Core i7, 4790I, 4Ghz, 8Mb de Cache;
- Memória DDR3 8GB;

4.1. Tempo de Execução

Para o primeiro teste realizado, verificou-se o comportamento do tempo de execução (em ms) de acordo com o tamanho dos grafos (de 5 a 14 vértices tanto para o número de voluntário quanto de focos) e densidades⁷ (0.3, 0.5, 0.7 e 0.9) dos grafos de voluntários e voluntários-focos. Foram realizados 10.000 testes contendo pares de grafos gerados aleatoriamente. Para cada densidade e tamanho de grafo, coletou-se a média de tempo de execução (em ms) para a solução ZikaZeroZ e um *Baseline* Força Bruta implementado. Os resultados são exibidos no Anexo I e nas Figuras 1 e 2.

⁷ A densidade de um grafo pode ser entendida como a probabilidade de uma aresta qualquer existir.

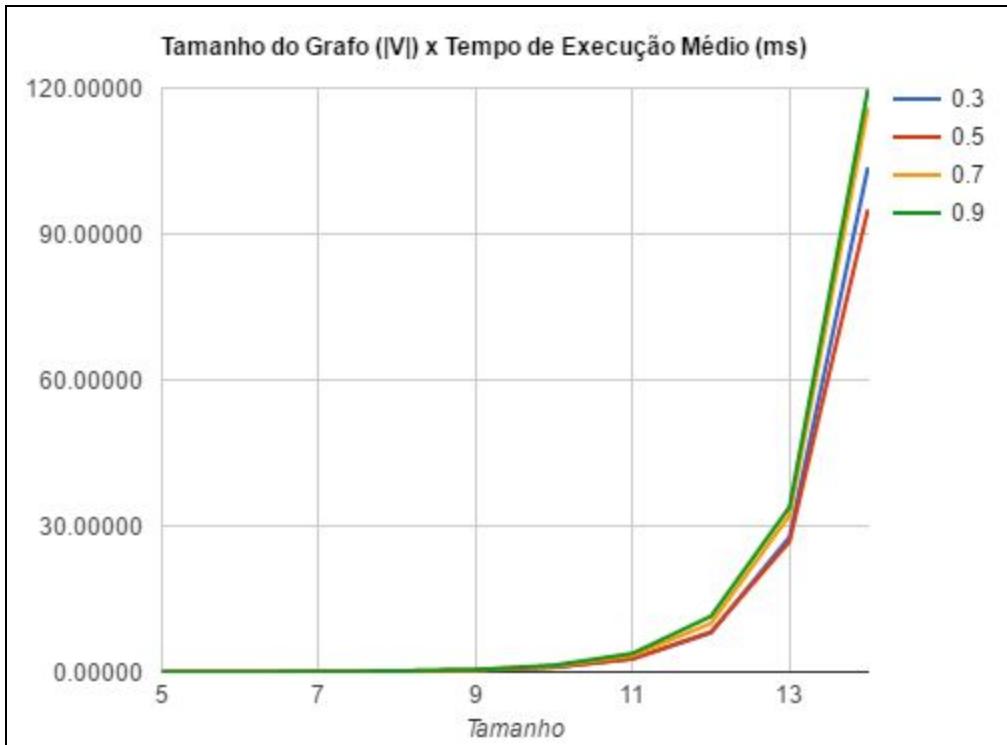
Figura 1: Desempenho do Algoritmo Implementado (ZikaZeroZ)



Da Figura 1 extrai-se que o algoritmo implementado apresenta um desempenho pior quanto maior o tamanho dos grafos de voluntários e de focos, algo que já era esperado, visto que a complexidade está relacionada diretamente com o tamanho dos mesmos. Além disso, constatou-se que o desempenho do algoritmo é muito pior para grafos pouco densos. Por exemplo, para um grafo de tamanho 12 e densidade 0.5, o tempo de execução médio foi de apenas 0.12ms, enquanto que para um grafo de densidade 0.3 do mesmo tamanho, o tempo de execução médio foi de 3ms, 24 vezes mais alto. Isso se deve ao fato de que, como há menos arestas, a probabilidade de se encontrar uma solução ótima em poucos níveis de recursão é mais baixa, ou seja, o algoritmo deve procurar por mais tempo e usar menos

heurísticas. De todo o modo, o algoritmo apresenta complexidade que se aproxima de uma função exponencial.⁸

Figura 2: Desempenho do Algoritmo Baseline (Força Bruta)



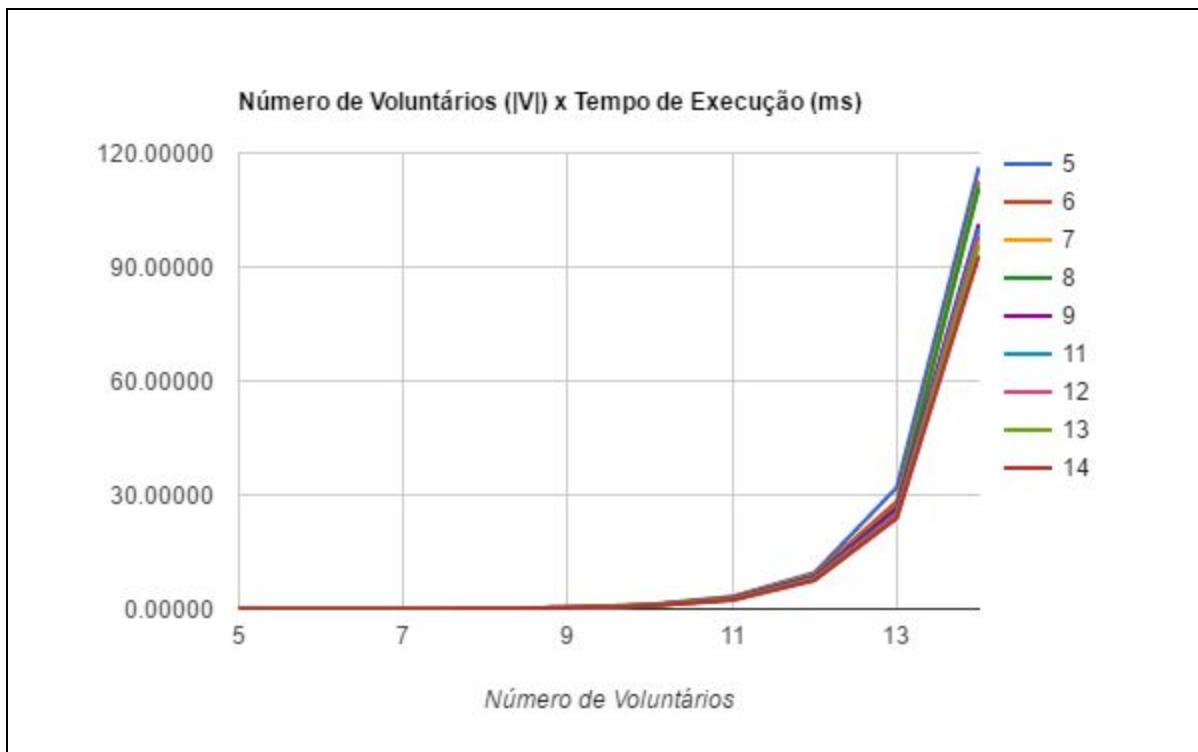
Assim, como no caso do algoritmo implementado, da Figura 2 extrai-se que o algoritmo força bruta apresenta um desempenho pior quanto maior o tamanho dos grafos de voluntários e de focos, algo que já era esperado pelos motivos já explicados. Além disso, constatou-se que o desempenho do algoritmo varia pouco em com diferentes densidades de grafos. Por exemplo, para um grafo de tamanho 12 e densidade 0.3, o tempo de execução médio foi de apenas 8ms, enquanto que para um grafo de densidade 0.9 do mesmo tamanho, o tempo de execução médio foi de 11ms, apenas 1.375 vezes mais alto. Ainda assim, constatou-se um pequeno aumento do tempo de execução quanto mais denso for o grafo. Isso decorre do fato de que o

⁸ Por exemplo, realizando uma regressão exponencial para curva de tempo de execução para grafos de densidade 0.3, foi encontrada a função $7.454900316 \cdot 10^{-5} e^{7.485257495 \cdot 10^{-1} |V|}$. Foi utilizado o website xuru.org.

grafo mais denso implicará em um maior número de testes e um maior número de soluções encontradas. O algoritmo força bruta só testará se a solução é ótima ao final do mesmo, ou seja, o grafo mais denso implicará em mais soluções e um maior número de testes após o procedimento recursivo. Assim como o algoritmo implementado, o algoritmo força bruta apresenta complexidade que se aproxima de uma função exponencial.⁹

O segundo teste realizado procurou analisar o tempo de execução para diferentes tamanhos de $|V|$ e $|F|$ mantendo-se uma mesma densidade nos grafos aleatórios gerados (0.5). Os resultados encontram-se disponíveis no Anexo II e nas Figuras 3 e 4.

Figura 3: Desempenho do Algoritmo Baseline (Força Bruta)

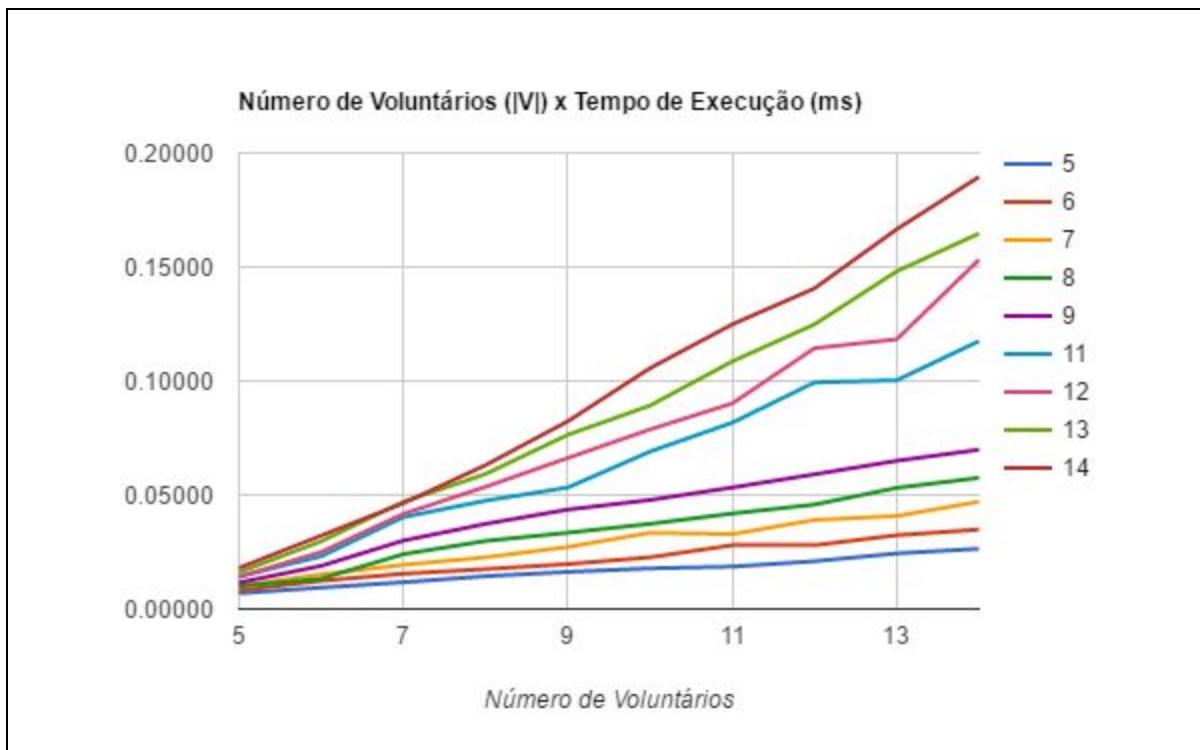


Apesar de difícil visualização, da Figura 3 extrai-se que o desempenho do algoritmo piora muito mais à medida que o número de voluntários aumenta do que quando o número de focos aumenta. Pelo contrário, observou-se uma ligeira melhora do algoritmo para valores maiores de $|F|$. Isso pode ser explicado pelo fato de que o algoritmo não implementa a heurística de *branch*

⁹ Por exemplo, realizando uma regressão exponencial para curva de tempo de execução para grafos de densidade 0.3, foi encontrada a função $t = 3.519781622 \cdot 10^{-6} e^{1.22712557 |V|}$. Foi utilizado o website xuru.org.

and *bound* e o desempenho está fortemente relacionado com o número de chamadas recursivas realizadas, o que depende apenas de $|V|$. A melhora do desempenho para valores maiores de $|F|$ deve-se ao fato de que são encontradas menos soluções proporcionalmente, reduzindo o trabalho de se remover as soluções não-mínimas ao término das recursões.

Figura 4: Desempenho do Algoritmo Implementado (ZikaZeroZ)



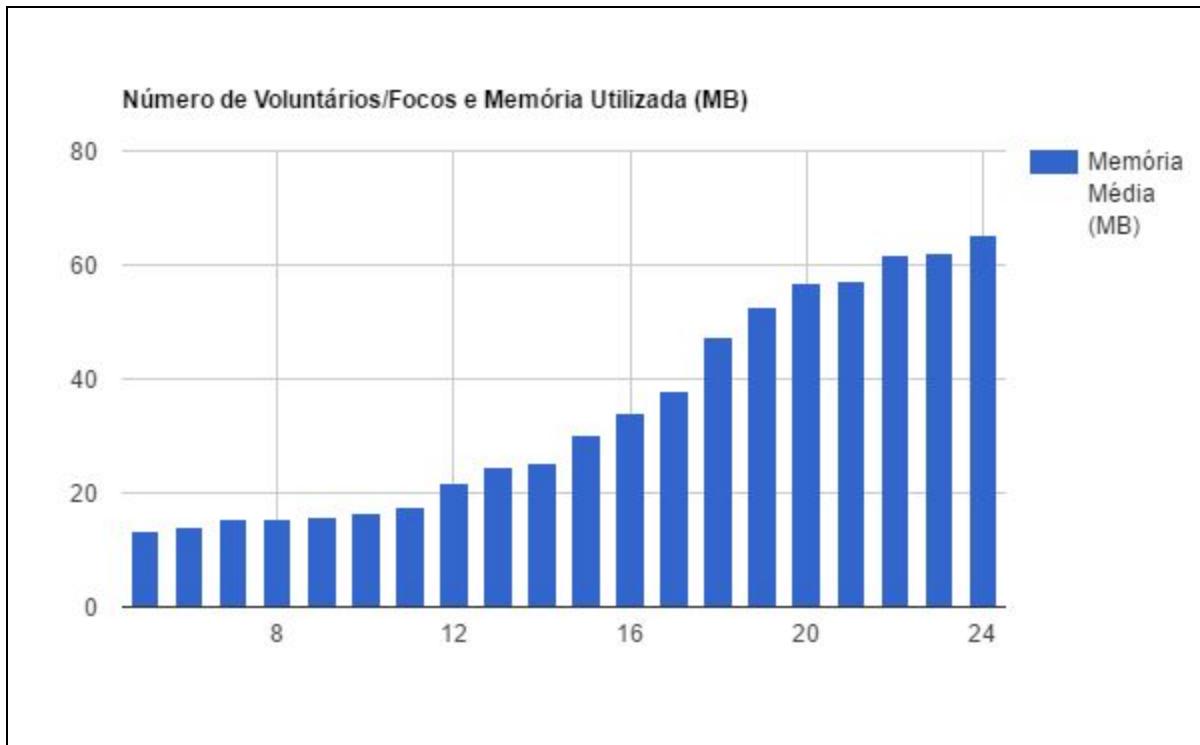
Da Figura 3, também extrai-se que o desempenho do algoritmo também piora muito mais à medida que o número de voluntários aumenta do que quando o número de focos aumenta. Isso se deve ao mesmo motivo do algoritmo força bruta, qual seja, o número de chamadas recursivas de **CobreFocos** está ligada ao valor de $|V|$ e não ao de $|F|$, conforme explicado na seção 3.2.1. Já a piora do desempenho quando o número de focos aumenta pode ser explicada pela maior demora de se encontrar uma solução mínima, visto que cada voluntário pode cobrir uma porcentagem fixa dos focos (nesse teste utilizou-se a densidade de 50%). Assim, quanto maior o número de focos, maior a probabilidade de se não cobrir todos os focos com um percentual de voluntários, sendo necessárias mais recursões.

Nota-se que o desempenho do algoritmo implementado foi muito superior ao do *baseline*, sendo milhares de vezes mais rápido para grafos com 14 voluntários. As heurísticas de *branch and bound* e o teste prévio de conectividade, a cada recursão, reduzem substancialmente o tempo de execução do algoritmo.

4.2. Memória

Para a realização do teste de utilização de memória, utilizou-se a classe Runtime¹⁰ e considerou-se a maior quantidade de memória utilizada durante a execução do programa. Os testes de utilização de memória foram realizados apenas sobre o algoritmo ZikaZeroZ e encontram-se na Figura 5. Assim como nos testes anteriores, foi considerada a média de utilização para 10.000 testes para cada valor de $|V|$ (igual a $|F|$), considerando-se a densidade igual a 0.5.

Figura 5: Utilização de Memória do Algoritmo Implementado (ZikaZeroZ)



¹⁰ Utilizou-se a expressão para um dado objeto rt de Runtime: `rt.totalMemory() - rt.freeMemory()`.

Da figura 5 percebe-se que a utilização de memória do algoritmo cresce à medida que os valores de $|V|$ e $|F|$ aumentam, conforme já demonstrado na seção anterior. Contudo, a taxa de crescimento foi muito inferior ao pior caso calculado (de ordem exponencial), aproximando-se de um valor polinomial de segunda ordem.¹¹

4.3. Corretude da Solução

Ressalta-se que todas as soluções encontradas para o algoritmo ZikaZeroZ foram as mesmas daquelas encontradas para o algoritmo *baseline* força bruta. Foram realizados testes usando grafos gerados aleatoriamente com densidades 0.3, 0.5, 0.7 e 0.9 e tamanhos 5 a 14. Para cada uma dessas configurações, foram gerados 10.000 grafos aleatórios e a solução foi exatamente a mesma para ambos os algoritmos. Para grafos de densidade média (0.5), foram encontradas soluções para grafos de até duas ordens de grandeza (mais de 100 voluntários e focos).

5. Conclusão

A implementação do procedimento ZikaZeroZ com heurísticas *branch and bound* promoveu uma substancial melhora do tempo de execução quando comparado com o algoritmo força bruta. Apesar de ser um problema NP-completo, de difícil resolução, a utilização de heurísticas gulosas e *branch and bound* aplica-se bem ao problema em tela. No entanto, o algoritmo implementado ainda possui complexidade fatorial, o que inviabiliza soluções para grafos muito grandes, mais de 1000 vértices por exemplo. Ainda assim, o algoritmo apresenta um desempenho aceitável, para o problema de seleção de uma rede colaborativa de voluntários que cobrem a totalidade de focos de dengue de uma cidade, dado o razoável tamanho dessa rede.

6. Referências

- [1] URRUTIA, Sebástian. Trabalho Prático: ZikaZeroZ.
- [2] Cormen, Thomas H. et al. (2009). Introduction to Algorithms, Third Edition. 3rd. The MIT Press. Isbn: 0262033844, 9780262033848.

¹¹ Utilizando-se a ferramenta de regressão (<http://www.xuru.org/rt/PR.asp>): $T = 1.218243213 \cdot 10^{-1} |V|^2 - 4.174492409 \cdot 10^{-1} |V| + 10.61568632$.

- [3] KANN, Viggo. *Minimum Hitting Set*. Disponível em <<https://www.nada.kth.se/~viggo/wwwcompendium/node149.html>>. Acesso em 27/04/2016.
- [3] Documentação Java: <https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html>

Anexo I - Primeira Bateria de Testes

Foram realizados conjuntos de 10.000 testes (variando-se a densidade e o tamanho dos grafos¹²) e coletou-se a média de tempo de execução (em ms) para a solução ZikaZeroZ e um *baseline* Força Bruta. Os resultados encontram-se na Tabela 1.

Tabela 1

Algoritmo	ZikaZeroZ	Baseline	ZikaZeroZ	Baseline	ZikaZeroZ	Baseline	ZikaZeroZ	Baseline
Densidade	0.30	0.30	0.50	0.50	0.70	0.70	0.90	0.90
Tamanho dos Grafos	Tempo Médio (ms)							
5	0.01094	0.00625	0.00469	0.01563	0.00297	0.01563	0.00113	0.01250
6	0.01406	0.01875	0.00625	0.02188	0.00438	0.02500	0.00117	0.02969
7	0.02188	0.02969	0.01094	0.05469	0.00531	0.07813	0.00117	0.07188
8	0.03594	0.15625	0.02188	0.13906	0.00797	0.20781	0.00160	0.19688
9	0.07969	0.35000	0.02656	0.35469	0.00953	0.45781	0.00168	0.46719
10	0.15781	0.94844	0.03438	1.00313	0.01578	1.20938	0.00219	1.37500
11	0.27813	2.70469	0.04688	2.69531	0.01750	3.39531	0.00234	3.80938
12	0.49219	8.07188	0.06563	8.23750	0.02203	10.02031	0.00270	11.46094
13	0.85625	27.73594	0.08281	26.80781	0.02656	32.18594	0.00305	33.89844
14	2.96875	103.69844	0.12188	95.05781	0.03047	116.01250	0.00355	119.74531

¹² Os grafos de voluntários e voluntários-focos possuem o mesmo tamanho.

Anexo II - Segunda Bateria de Testes

Foram realizados conjuntos 10.000 testes (variando-se o tamanho dos grafos, isso é, valores de $|V|$ e $|F|$, mantendo-se a densidade igual a 0.5) e coletou-se a média de tempo de execução (em ms) para a solução ZikaZeroZ e um *baseline* Força Bruta. Os resultados encontram-se na Tabela 2.

Tabela 2

(V)	(F)									
ZikaZeroZ	5	6	7	8	9	10	11	12	13	14
5	0.01094	0.00547	0.00703	0.00469	0.00391	0.00469	0.00547	0.00313	0.00469	0.00625
6	0.02188	0.01719	0.01406	0.01719	0.01172	0.01563	0.01406	0.01406	0.01016	0.01328
7	0.04531	0.04297	0.03750	0.03906	0.03438	0.03516	0.03281	0.03438	0.03047	0.03047
8	0.14063	0.12969	0.12188	0.11406	0.10547	0.10469	0.10234	0.09844	0.09375	0.08672
9	0.40234	0.38203	0.37734	0.38906	0.33906	0.32656	0.32031	0.30391	0.29063	0.29219
10	1.16563	1.06016	1.05000	1.00156	0.94063	0.92813	0.90625	0.87188	0.84844	0.81406
11	3.22188	3.08906	3.00938	2.96719	2.83359	2.67031	2.56172	2.56797	2.40703	2.30234
12	9.53594	9.17031	8.91641	8.66719	8.53984	8.56563	8.14766	7.85938	7.73047	7.49453
13	31.96875	28.26563	27.10938	26.92188	26.43750	25.50000	25.18750	24.81250	24.01563	23.89063
14	116.328125	112.515625	111.546875	111.046875	101.390625	100.890625	99.90625	98.140625	95.890625	93.078125
Baseline	5	6	7	8	9	10	11	12	13	14
5	0.00688	0.00813	0.01016	0.01000	0.01156	0.01203	0.01422	0.01406	0.01609	0.01797
6	0.00938	0.01234	0.01516	0.01328	0.01891	0.02172	0.02313	0.02500	0.02969	0.03219
7	0.01172	0.01547	0.01938	0.02406	0.03000	0.03063	0.04031	0.04172	0.04703	0.04641
8	0.01438	0.01750	0.02266	0.02984	0.03734	0.03938	0.04750	0.05359	0.05922	0.06313
9	0.01625	0.01969	0.02719	0.03344	0.04359	0.04875	0.05328	0.06625	0.07641	0.08234
10	0.01781	0.02266	0.03344	0.03734	0.04781	0.05406	0.06906	0.07875	0.08922	0.10563
11	0.01859	0.02797	0.03281	0.04188	0.05328	0.06844	0.08172	0.09016	0.10859	0.12484
12	0.02094	0.02797	0.03906	0.04578	0.05906	0.07656	0.09938	0.11438	0.12484	0.14063
13	0.02438	0.03234	0.04078	0.05313	0.06500	0.08438	0.10031	0.11828	0.14813	0.16656
14	0.02641	0.03484	0.04703	0.05750	0.06984	0.09406	0.11750	0.15313	0.16469	0.18953

BRUNO LUAN DE SOUSA

RELATÓRIO DO TRABALHO PRÁTICO

**BELO HORIZONTE
MINAS GERAIS – BRASIL
2016**

Exercício 1 - Apresente uma modelagem para o Problema ZikaZeroZ empregando grafos. Deixe claro quais as restrições e/ou suposições devem ser feitas.

A modelagem do problema do ZikaZeroZ em grafos consiste de uma solução que se baseia basicamente em dois conjuntos de vértices, onde no primeiro os vértices conectam entre si, e com vértices do outro conjunto, permitindo que o grafo permaneça conexo.

A partir disso, utilizam-se dois tipos de vértices e dois tipos de arestas para a representação e modelagem do problema. O primeiro tipo de vértice, ou vértice primário, é responsável por representar os indivíduos voluntários e integra o primeiro conjunto do grafo. O segundo tipo de vértice, ou vértice secundário, é responsável por representar os focos de reprodução do mosquito, e integra o segundo conjunto do grafo.

Após estabelecer a representação dos vértices, será mostrado a representação das arestas. O primeiro tipo de aresta, ou aresta primária, é responsável por representar a relação de amizade entre dois indivíduos voluntários, que consiste na conexão de dois vértices primários localizados no primeiro conjunto. Essa aresta é não orientada, assim, em uma conexão entre dois vértices u e v , u é adjacente de v , e v é adjacente de u . O segundo tipo de aresta, ou aresta secundária, é responsável por representar a cobertura de um foco por um determinado indivíduo, que consiste na conexão de um vértice primário com um vértice secundário. Além disso, a aresta secundária conecta o primeiro conjunto com o segundo e garante que grafo permaneça conexo. Essa aresta é orientada, e apenas os vértices primários (indivíduos voluntários) tem os vértices secundários (focos) como seus adjacentes.

A **Figura 1** ilustra a modelagem do problema ZikaZeroZ na forma de um grafo.

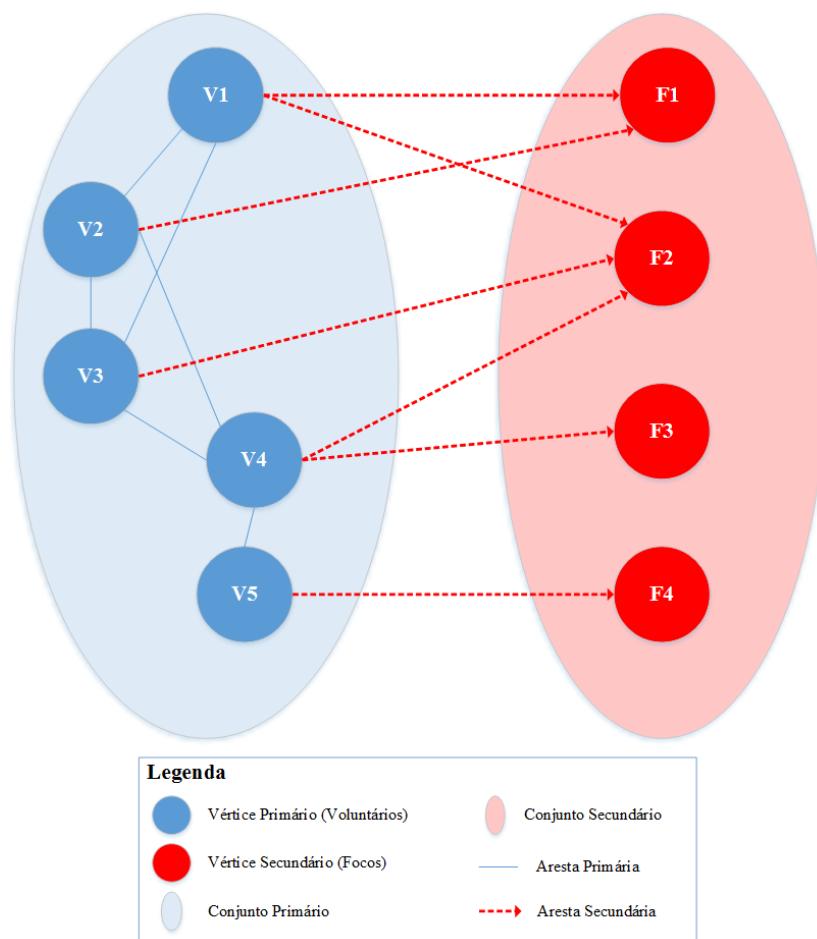


Figura 1. Representação da modelagem do problema ZikaZeroZ em grafos.

Após a modelagem do problema ZikaZeroZ na representação de grafos, como mostrado na **Figura 1**, modelou-se sua estrutura de representação em nível de código que está ilustrada por um diagrama de classes na **Figura 2**.

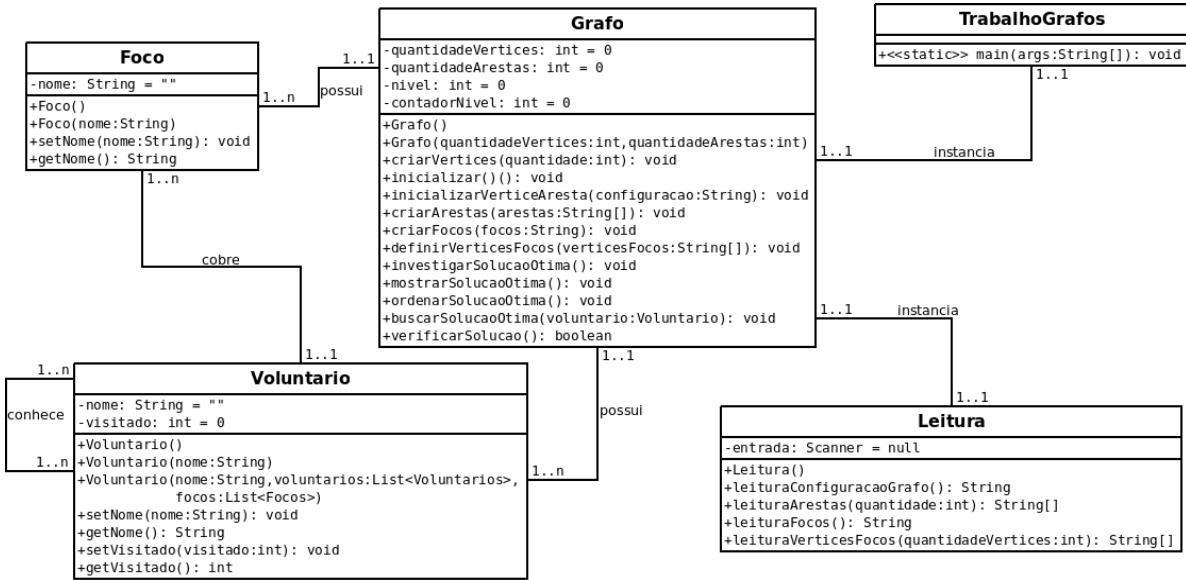


Figura 2. Estrutura de representação do problema ZikaZeroZ em nível de código.

Exercício 2 - Descreva um algoritmo para resolver o Problema ZikaZeroZ, tendo em vista a modelagem realizada no Exercício 1.

Pseudocódigo do algoritmo

```

1   flag = false
2   nivel = 0
3   contadorNivel = 0
4   solucaoOtima = null //é formada por uma lista de voluntários candidatos a solução ótima.
5
6   inicializa o grafo G
7   function investigarSolucaoOtima(){
8       enquanto nivel <= à quantidade de voluntarios existente no grafo && flag == false{
9           for cada voluntario v ∈ G{
10               contadorNivel = 0
11               buscarSolucaoOtima(v)
12               if(flag == verdadeiro){
13                   saí do laço de repetição for
14               }
15           }
16           incrementa a variável nivel em 1
17       }
18       mostraSolucaoOtima()

```

```

19    }
20
21
22 function buscarSolucaoOtima(Voluntario voluntario) {
23     marca voluntario como visitado
24     if (contadorNivel == nivel) {
25         adiciona voluntario na lista de solucao otima.
26         if (verificarSolucao()) {
27             flag = true
28         } else {
29             marca voluntario com não visitado
30             remove voluntario da solucaoOtima
31             decrementa a variável contadorNivel em 1
32         }
33     } else {
34         if (!flag) {
35             voluntariosAdjacentes = adjacentes de voluntario;
36             adiciona voluntario à solucaoOtima;
37             if (adjacentes de voluntario ==1 && esse adjacente já foi
visitado) {
38                 buscarSolucaoOtima(adjacente de voluntario);
39                 if (flag == false) {
40                     marca voluntario como não visitado
41                     remove voluntario da solucaoOtima
42                 }
43             } else {
44                 for cada v ∈ voluntariosAdjacentes{
45                     if (v não foi visitado ainda) {
46                         incrementa contadorNivel;
47                         buscarSolucaoOtima(v);
48                     }
49                 }
50                 if (flag == false) {
51                     marca voluntario como não visitado
52                     remove voluntario da solucaoOtima
53                     decrementa contadorNivel
54                 }
55             }
56         }
57     }
58 }
```

O algoritmo mostrado acima é uma adaptação da busca em profundidade iterativa, onde ele faz combinações dos vértices de voluntários de acordo com o nível estabelecido pela variável nível (linha 2). A variável contadorNivel (linha 3) é responsável por garantir que a lista de solução ótima sempre contenha combinações de voluntários respeitando o nível estabelecido. A variável solucaoOtima (linha 4) contém combinações de voluntários candidatos a serem a solução ótima do problema. A variável flag (linha 1) é utilizado como um critério de parada para o algoritmo, quando este encontra a primeira solução ótima para

o problema.

A partir disso, ao começar seu funcionamento, a primeira ação que este vai exercer é inicializar o grafo G (linha 6) e estabelecer os voluntários, focos, relações entre voluntários e relações entre voluntários e focos. Após realizar essa ação, o algoritmo começa a execução da função investigarSolucaoOtima (linha 7). Essa função é responsável por realizar apenas o aumento do nível de busca da solução ótima (linha 16) e garantir que para cada nível de busca seja iniciada a busca de cada um dos voluntários presentes no Grafo (linhas 9 – 15). Na linha 11 é realizada uma chamada da função buscarSolucaoOtima e passado o voluntário atual como parâmetro. A função buscarSolucaoOtima é responsável por fazer as combinações dos voluntários na variável solucaoOtima respeitando sempre o nível estabelecido, e verificar se a solução atual presente na variável solucaoOtima é realmente a melhor solução para o Algoritmo.

Ao começar seu funcionamento, a primeira ação da função buscarSolucaoOtima é marcar o voluntário recebido como visitado (linha 23) para que ele não seja visitado novamente. Na linha 24 é comparado se a variável contadorNivel possui o mesmo valor que a variável nivel. Caso essa solução seja satisfeita, as linhas 25 – 32 vão adicionar o voluntário à lista de solução ótima e será realizado a verificação da solução, se esta cobre todos os focos. Caso ela cubra, a flag é setada para true. Caso ela não seja, o voluntário é remarcado, desta vez como visitado, é removido da lista de solução ótima e o nível atual indicado na variável contadorNivel é reajustado.

Caso a comparação realizada na linha 24 não seja satisfeita, as linhas 34 – 56 utiliza a recursividade para percorrer os voluntários conectados, e adicionar sempre os voluntários não visitados na lista de solução.

Quando uma solução ótima é encontrada, a variável flag é marcado como true, e o algoritmo desfaz toda a recursividade na função buscarSolucaoOtima. Ao retornar à função investigarSolucaoOtima, o algoritmo sai do laço de repetição enquanto e exibe a solução ótima para usuário.

Exercício 3 - Analise as complexidades Temporais e Espaciais (usando Nota c̄ao Assintótica) do algoritmo proposto no Exercício 2.

Análise Temporal

Como mostrado no exercício anterior, o algoritmo tem duas funções básicas que são utilizadas para poder realizar a busca da solução ótima. Assim, realizando a análise das funções mostradas no pseudocódigo acima, tem-se o seguinte:

- A função investigarSolucaoOtima tem como sua principal operação, a chamada da função buscarSoluçãoOtima. Para realizar essa chamada, é percorrido um laço externo n vezes, onde n representa a profundidade no grafo que está sendo investigado. Ainda, existe um laço interno que percorre a quantidade total de voluntários do grafo (v). Assim, pode-se dizer que a função de complexidade temporal é: $T(n) = n*v$.
- A função buscar SolucaoOtima tem como sua principal operação a chamada recursividade dela mesma passando como parâmetro um vértice adjacente do vértice atual. Como a recursividade é realizada até todas as combinações de vértices do nível indicado seja satisfeita, pode-se dizer que a função de complexidade temporal é: $T(n) = n!*v$.
- Ainda, existe um ponto importante dentro da função buscarSolucaoOtima, que é a chamada da função verificarSolucaoOtima. Essa função verificarSolucaoOtima,

percorre toda a lista da solução ótima, comparando a conexão de cada voluntário dessa lista com cada foco existente no grafo para saber se a solução consegue cobrir todos os focos. Assim, como a lista de solução ótima é de tamanho n e a lista de foco é de tamanho f , pode-se dizer que a função de complexidade temporal é: $T(n) = n*f$.

•Juntando as três funções de complexidade, tem-se que a função de complexidade temporal total do algoritmo é: $T(n) = n!*v*f$. Assim, a complexidade desse algoritmo é $O(n!)$.

Análise Espacial

Como mostrado no exercício anterior, o algoritmo tem duas funções básicas que são utilizadas para poder realizar a busca da solução ótima. Assim, realizando a análise das funções mostradas no pseudocódigo acima, tem-se o seguinte:

•A função investigarSolucaoOtima tem como sua principal operação, a chamada da função buscarSoluçãoOtima. Para cada chamada realizada é gasto um espaço de memória para que a função realize a sua execução. Como é percorrido um laço externo n vezes, onde n representa a profundidade no grafo que está sendo investigado e depois um laço interno que percorre a quantidade total de voluntários do grafo (v), pode-se dizer que a função de complexidade espacial é: $S(n) = n*v$.

•A função buscarSoluçãoOtima tem como sua principal operação a chamada recursividade dela mesma passando como parâmetro um vértice adjacente do vértice atual. Para cada chamada recursiva que é feita, é realizado um empilhamento de chamada, e como a recursividade é realizada até todas as combinações de vértices do nível indicado seja satisfeita, pode-se dizer que a função de complexidade espacial é: $S(n) = n!*v$.

•Ainda, existe um ponto importante dentro da função buscarSoluçãoOtima, que é a chamada da função verificarSoluçãoOtima. Essa função verificarSoluçãoOtima, percorre toda a lista da solução ótima, comparando a conexão de cada voluntário dessa lista com cada foco existente no grafo para saber se a solução consegue cobrir todos os focos. Assim, para cada comparação realizada nessa função é gasto um espaço de memória. Como a lista de solução ótima é de tamanho n e a lista de foco é de tamanho f , pode-se dizer que a função de complexidade espacial é: $S(n) = n*f$.

•Juntando as três funções de complexidade, tem-se que a função de complexidade espacial total do algoritmo é: $S(n) = n!*v*f$. Assim, a complexidade espacial desse algoritmo é $O(n!)$.

Uma observação que deve ser feita é que no algoritmo o n , mostrado na análise da complexidade acima, é representado pela variável nível que é iniciada sempre com o valor 0. Porém, esse valor inicial da variável nível representa na verdade que será analisado a profundidade 1 de cada um dos vértices do grafo (apenas o próprio vértice) para verificar algum desses vértices são a solução ótima. Para o valor 1 da variável nível, a profundidade a ser pesquisada é 2, e assim por diante.

Portanto, o n mostrado na análise de complexidade acima, chamado de profundidade, será o valor da variável nível do algoritmo acrescida de 1, ou seja, a quantidade de vértices compõe a solução ótima final.

Exercício 4 - Implemente o algoritmo proposto no Exercício 2 na linguagem de programação C, C++, Java ou Python.

A implementação do algoritmo proposto no Exercício 2 foi realizado na linguagem Java contendo 5 classes: Foco, Grafo, Leitura, TrabalhoGrafos (onde encontra-se o método main que inicializa o algoritmo) e Voluntário. As classes e o código de implementação encontram-se junto a esse relatório, compactados em no arquivo .zip enviado através do sistema minhaUFMG.

Exercício 5 - Execute testes da implementação do Exercício 4, propondo instâncias interessantes para o Problema ZikaZeroZ. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n , m e r). Outra sugestão é relatar o tamanho da maior instância para o qual foi produzida uma solução.

Para realização dos testes, foram utilizadas seis instâncias de diferentes tipos, e medida a execução nas seguintes operações do Algoritmo: chamadas da função buscarSolucaoOtima (tanto recursivas quanto por outras funções) e a comparação da lista auxiliar de focos (comparação situada dentro dos laços de repetições) na função verificarSolucaoOtima. Foi utilizado um contador (encontra-se comentado no código fonte do algoritmo) para verificar quantas vezes essas operações eram chamadas até encontrar a solução ótima e foram obtidos os seguintes resultados mostrado gráfico abaixo.

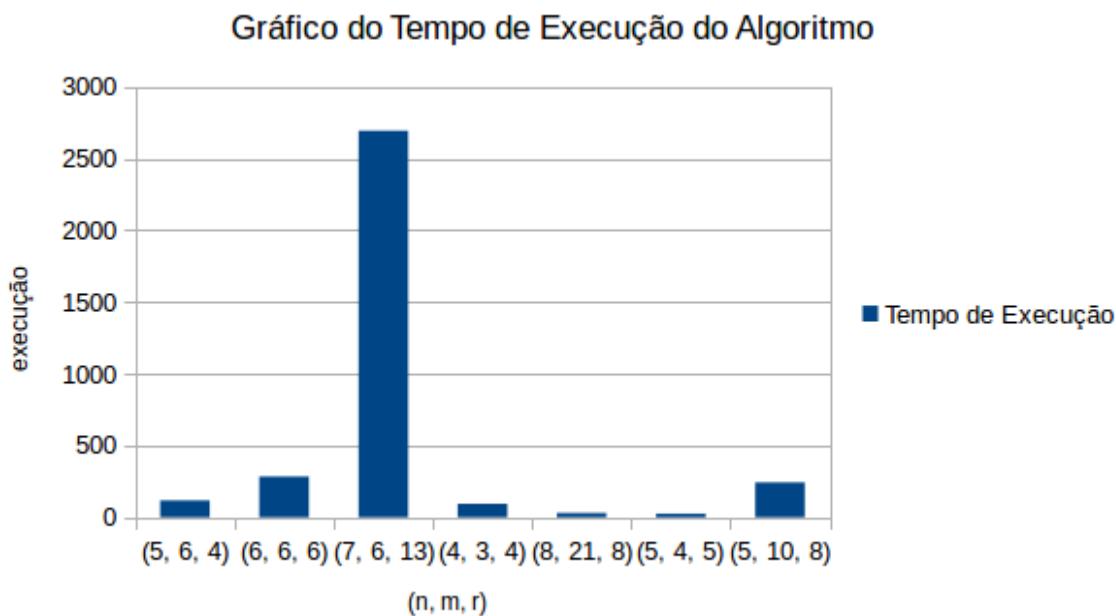


Figura 3. Gráfico de Tempo da Execução do Algoritmo

O gráfico da **Figura 3**, levou em consideração dois fatores: a entrada expressa em (n, m, r) , onde n representa o número de voluntários, m representa o número de relações entre os voluntários e r representa o número de focos, e o tempo de execução expresso em número de vezes que o algoritmo executou a operações em análise. A entrada está representada pelo eixo x e o tempo de execução está representado pelo eixo y.

Foram realizados testes com seis instâncias diferentes, como mostrados no gráfico. A primeira instância resultou no valor de 117 execuções de operações analisadas no algoritmo. A segunda instância resultou no valor de 284 execuções. A terceira instância resultou no valor de 2692 execuções. A quarta instância resultou no valor de 94 execuções. A quinta instância resultou no valor de 30 execuções. A sexta instância resultou no valor

de 25 execuções. E por fim, a sétima instância resultou no valor de 242 execuções.

Uma informação importante obtida após a execução dos testes é que o tempo de execução do algoritmo, número de vezes que o algoritmo executa uma operação em análise, é diretamente influenciada pela profundidade da resposta (número de vértices que compõe a solução ótima final). Assim, quanto maior for a profundidade da solução ótima final, maior será o tempo de execução do algoritmo para encontrar a solução ótima.

Exercício 6 - Compare a análise e a execução, respectivamente, Exercícios 3 e 5. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

Após realizar os testes no algoritmo, exercício 5, percebeu-se que de acordo com que os focos estão conectados entre os voluntários e de acordo com que os voluntários estão conectados entre si, o algoritmo pode ter um melhor desempenho, executar mais rápido, ou ter um pior desempenho, demorar mais para encontrar a solução. Baseado nisso, ao analisar o algoritmo no exercício 2, percebeu-se que a função de complexidade do algoritmo era de $f(n) = n! * v * f$.

Ao comparar os resultados obtidos pela função de complexidade e os resultados reais obtidos pelo algoritmo, para algumas instâncias percebeu-se que houve uma diferença. A principal explicação para essa diferença de valores, deve-se justamente a configuração dos focos e a configuração das relações entre voluntários, pois, quando o algoritmo encontra a solução ótima, ele simplesmente desfaz toda a recursividade feita até o momento e termina o algoritmo. Ele não faz todas as combinações possíveis para o respectivo nível em que está atuando, o que faz com que aconteça essa diferença de valores.

Trabalho Prático - Grafos

Renato Florentino Garcia
2016661865

4 de maio de 2016

UFMG
DCC865 PG Projeto e Análise de Algoritmos
For: Prof. Sebastián Urrutia

Exercício 1

Dado um grafo $G(V, A)$, onde $V = \{v_1, v_2, \dots, v_n\}$ é um conjunto de n vértices e $A = \{a_1, a_2, \dots, a_m\}$ é um conjunto de m arestas; e um conjunto de focos $F = \{f_1, f_2, \dots, f_r\}$. Seja também $R : V \rightarrow \mathcal{P}(F)$ uma relação que mapeia vértices para um conjunto de i focos, onde i pode ser qualquer valor entre 0 e r ($\mathcal{P}(F)$ é o conjunto potência de F).

O problema ZikaZeroZ se resume portanto a encontrar um subconjunto $V_z \subset V$ com a menor cardinalidade possível, tal que a função R aplicada à V_z acesse todos os focos f_i , e V_z induza em G um subgrafo conexo $G_z(V_z, A_z)$.

Exercício 2

Data: $G(V, A), F, R$
Result: V_z

```
1 foreach  $V_p \in \mathcal{P}(V)$  dados em ordem não-decrescente de cardinalidade do
2   | if  $R(V_p)$  contém todos os focos  $f_i$  e o grafo  $G_z$  induzido é conexo then
3   |   | return  $V_p$ 
4   | end
5 end
```

Algoritmo 1: ZicaZeroZ

Data: $G(V, A), V_z$
Result: True se o grafo induzido em G por V_z é conexo. False caso contrário.

```
1  $Q \leftarrow (v_{z0});$ 
2  $V_z \leftarrow V_z \setminus \{v_{z0}\};$ 
3 while  $Q \neq \emptyset$  do
4   |  $u, Q \leftarrow (q_0) \parallel (q_1 \dots q_n);$ 
5   | foreach  $v \in \{v_i : (u, v_i) \in A\}$  do
6   |   | if  $v \in V_z$  then
7   |   |   |  $V_z \leftarrow V_z \setminus \{v\};$ 
8   |   |   |  $Q \leftarrow Q \parallel (v);$ 
9   |   | end
10  | end
11 end
12 return True se  $V_z = \emptyset$ , False caso contrário.
```

Algoritmo 2: Teste conexo

Exercício 3

Complexidade Espacial

A estrutura de dados usada para armazenar o grafo G é uma lista de adjacências, logo a memória utilizada será $\Theta(V + A)$. Todas as listas, V_z, Q , armazenarão no máximo $|V|$ itens, e portanto são $\Theta(V)$. A função R é armazenada usando a mesma estrutura que a lista de adjacências e portante é $\Theta(V + F)$. Como em geral o número de focos é menor ou próximo ao número de vértices, o algoritmo ZicaZeroZ como um todo é $\Theta(V + A)$.

Complexidade Temporal

No algoritmo 1, o loop na linha 1 será executado $2^{|V|}$ vezes, portando $\Theta(V!)$. A linha 2 fará dois testes. No primeiro teste é verificado se $R(V_p)$ contém todos os focos em F e sua execução é $\Theta(V_p) = \Theta(V)$. O segundo teste verifica se o subgrafo induzido G_z é conexo, e está descrito no algoritmo 2.

No algoritmo 2, o loop da linha 3 será executado no pior caso uma vez para cada elemento em V_z , portanto $\Theta(V_p) = \Theta(V)$. O loop da linha 5 será executado uma vez para cada aresta ligada ao vértice u , sendo portanto $\Theta(A)$. Com isso, o algoritmo 2 ao todo será $\Theta(VA)$.

Combinando os resultados encontrados nos parágrafos anteriores, o algoritmo 1 como um todo será $\Theta(V!)$.

Exercício 4

Uma implementação dos algoritmos propostos escrito na linguagem Python está anexa à entrega deste relatório.

Exercício 5

Para testar o programa implementado no Exercício 4, foi desenvolvido um segundo programa que recebe como entrada um número de vértices n , um número de arestas m , e um número de focos r ; e produz como saída um grafo $G(V, A)$ e uma função R , ambos aleatórios e descritos como no formato de entrada definido para este trabalho prático. Todos os $G(V, A)$ gerados são necessariamente conexos, e toda função R tem como imagem um conjunto com todos os $f_i \in F$. Com estas restrições é garantida a existência de uma solução para o presente problema em G e F .

O gráfico visto na figura 1 mostra os tempos de execução do algoritmo 1 como implementado no exercício 4. Para cada quantidade de vértices no eixo x , foram gerados 10 grafos aleatórios, com cada grafo tendo n vértices, $(n^2 - n)/4$ arestas e $n/2$ focos. O tempo em segundos no eixo y é o tempo médio de execução do algoritmo para os 10 grafos.

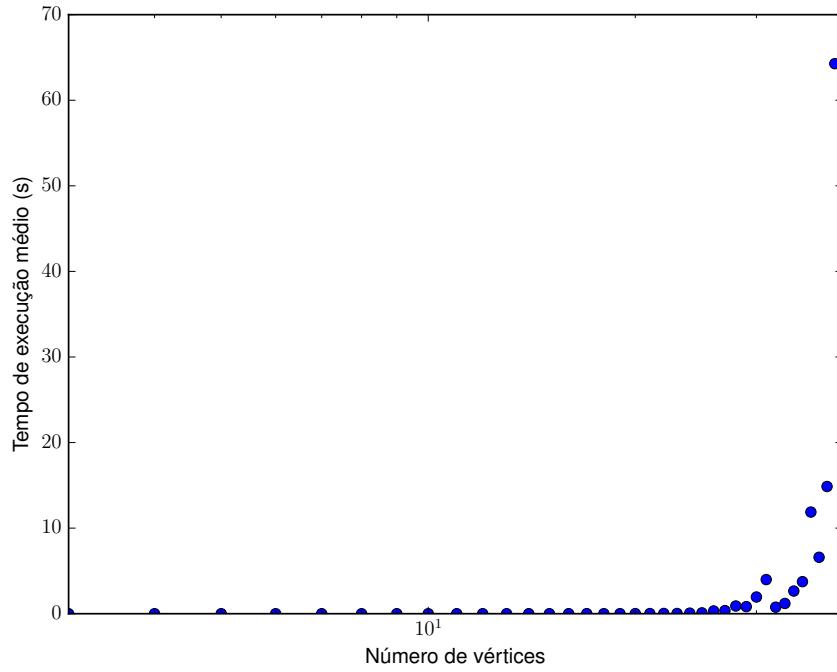


Figura 1: Tempos de execução do algoritmo implementado.

Exercício 6

A fim de se comparar o tempo de execução medido durante a execução do algoritmo implementado com o resultado teórico levantado durante a análise do mesmo, foram feitos dois ajustes de curva aos dados mostrados no gráfico da figura 1. A primeira função ajustada foi $f_1(n) = a_1 2^n$, que é o número de vezes que o loop principal do algoritmo seria executado. A segunda função ajustada foi $f_2(n) = a_2 n!$, que é a complexidade temporal do algoritmo. Em ambas funções n é o número de vértices, e a_1 , a_2 são constantes para escalar as funções de acordo com os valores em segundos.

Os ajustes obtidos podem ser observado nos gráficos da figura 2, onde as constantes $a_1 = 1.026 \cdot 10^{-10}$ e $a_2 = 3.168 \cdot 10^{-45}$. Como pode ser observado, o algoritmo implementado se comportou de acordo com o previsto pela análise teórica, sendo possível ajustar as curvas das duas funções f_1 e f_2 somente escolhendo uma constante multiplicativa, o não altera o comportamento assintótico das mesmas.

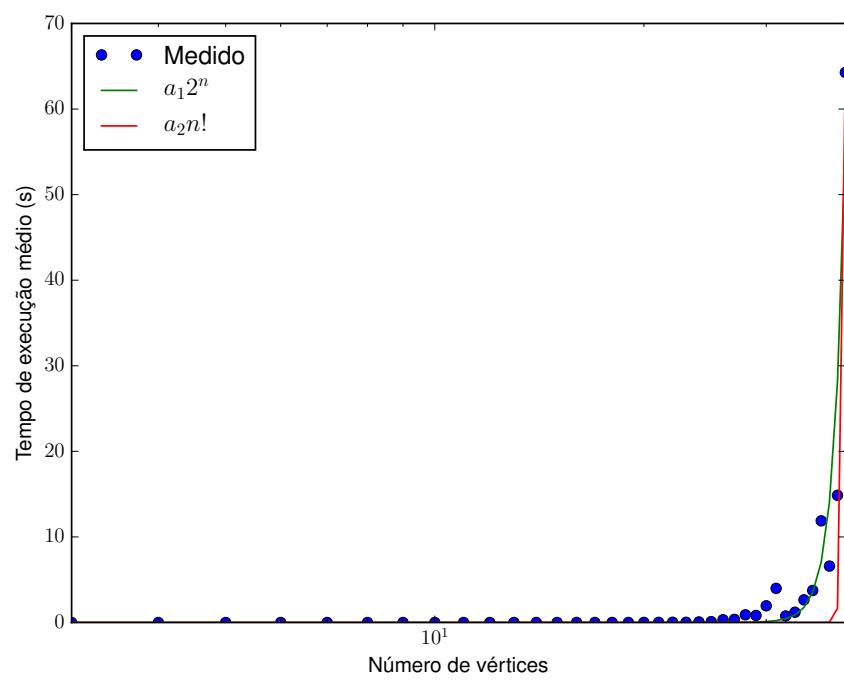


Figura 2: Funções de aproximação ao tempo de execução.

Documentação para o Trabalho de Projeto e Análise de Algoritmos

Ricardo Barbosa Kloss
UFMG Computer Science Department
Belo Horizonte, Brazil

2016 - 01

1 Exercício 1 - Modelagem

O problema consiste de duas relações, uma de voluntários para voluntários, e outra de voluntários para focos de zika. O que se procura é um subconjunto de voluntários tal que a partir desse subconjunto se tenha acesso a todos os focos de zika e que o grafo induzido gerado por esse subconjunto seja conexo, isto é, dado qualquer dois voluntários desse subconjunto é possível encontrar um caminho de um para outro sem passar por um nó ausente no subconjunto.

Para resolver o problema modelamos a relação de amizade dos voluntários como um grafo $n \times m$, onde temos n voluntários e m arestas, onde o voluntário i tem aresta com o voluntário j caso eles sejam amigos.

A relação entre voluntários e focos de zika pode ser modelada como um grafo bipartido que relaciona o conjunto de voluntários com o conjunto de focos de zika.

A quantidade de subconjuntos possíveis é 2^n aonde n é a quantidade de voluntários. Isto porque cada voluntário pode ou não estar no subconjunto. Cada subconjunto que satisfaz a condição de que o grafo induzido gerado por ele é conexo é uma solução do problema, mas, queremos a menor solução possível.

1.1 Restrições da Solução

Para que um subconjunto de voluntários seja solução ele deve satisfazer duas condições. O subgrafo induzido deve ser conexo e todos os focos devem ser alcançáveis a partir do subconjunto de voluntários selecionados.

Em seguir explicamos em detalhe cada uma dessas restrições.

1.1.1 Conectividade do Grafo

Para verificar se o grafo é conexo podemos fazer uma busca em profundidade partindo de um nó da possível solução, se um nó da possível solução não for visitado ao final da busca, então o grafo induzido não é conexo.

1.1.2 Cobertura de Focos

Uma das restrições que uma solução deve satisfazer é que todos os focos sejam alcançáveis a partir de algum voluntário selecionado. Para verificar isto podemos iterar nos nós da solução e marcar os focos que eles tem acesso, se todos focos forem marcados a solução é viável na cobertura.

2 Exercício 2 - Algoritmo

Nesta seção explicamos as duas abordagens realizadas para achar a menor solução para o problema. A primeira subseção 2.1 fala da busca em largura no espaço de possibilidades, a segunda subseção 2.2 fala da busca em profundidade no espaço de conjuntos de voluntários de cada foco.

Para um conjunto de voluntários de tamanho n temos um total de 2^n possíveis soluções.

2.1 Busca em Largura

O algoritmo proposto nessa seção cria uma árvore de possibilidades partindo da solução vazia ($S = \emptyset$), sem nenhum voluntário, como nó raiz. Para encontrar as soluções de menor tamanho realizamos uma busca em largura nessa árvore implícita e verificamos quais sub conjuntos de voluntários formam uma solução satisfatória. Por utilizarmos uma busca em largura, só será buscada uma solução de tamanho $i + 1$ após ter se garantido que não existe nenhuma solução de tamanho i .

Para evitar que nós já explorados fossem explorados novamente, foi feito uso de um conjunto de estados explorados, chamado de fronteira. Utilizamos uma fronteira só, ela guarda os estados do próximo nível que já foram olhados, mas, não explorados, dessa forma, não são inseridos estados repetidos na fila de exploração. A inspiração para essa solução foi retirada da memoização de programação dinâmica e da busca em uma árvore de possibilidades utilizada em inteligência artificial. Um problema dessa solução é que a fronteira pode ter um custo em espaço muito alto. Outro problema é que em cada nó devemos verificar se o subconjunto de voluntários é solução.

2.2 Busca em Profundidade

Em outro algoritmo proposto para resolver o problema também criamos uma árvore implícita, mas, dessa vez cada nível representa os voluntários que se

conectam ao respectivo foco, isto é, os nós no nível i são relativos aos nós que tem aresta com o foco i . Essa árvore tem profundidade igual ao número de focos, f . Ao se percorrer em profundidade a árvore temos que em cada nó está sendo adicionado um voluntário ao subconjunto solução e também sabemos que o subconjunto solução de um nó folha está garantido de cobrir todos os focos, aonde só resta testar se ele é conexo. Devido a essa última propriedade, se fizermos a busca em profundidade na árvore não precisamos de avaliar se um candidato a solução cobre todos os focos, pois isto está garantido, assim, só é necessário checar, cada vez que se chega a um nó folha, se o subconjunto de voluntários relativo aquele nó é um subconjunto que gera um subgrafo induzido conexo. Um possível problema dessa abordagem é que ao contrário da busca em largura, quando uma solução é achada não temos prova de que essa é a menor solução, assim, é necessário continuar percorrendo a árvore, entretanto, podemos podar os nós cujo subconjunto de voluntários é maior que o tamanho da menor solução encontrada.

3 Exercício 3 - Análise de Complexidade

Os cálculos de complexidade a seguir levam em consideração que o grafo de amizades possui n voluntários, f focos, e_v amizades e e_f relações entre voluntários e focos.

3.1 Complexidade da Solução por Busca em Largura

A complexidade da solução no melhor caso é $\Theta(n(n + m + f + e_f))$ pois a menor solução tem tamanho 1 e portanto basta olhar n casos, os singletons formados por cada voluntário, e verificar se são soluções. No pior caso a menor solução tem tamanho $\Theta(n)$ e para verificar isso é necessário olhar todas 2^n outras possibilidades para se assegurar que não há solução de menor tamanho.

Para verificar se uma combinação de voluntários é solução fazemos uma busca em profundidade e uma busca no grafo de foco de zika cada um tem complexidade de $\Theta(n)$, usando uma matriz de adjacência. Assim a complexidade final é $\Theta(n(n + m + f + e_f))$ no melhor caso e $\Theta((2^n)(n + m + f + e_f))$ no pior caso.

Como o grafo de amizades inicial é sempre conexo, não há necessidade de buscar suas componentes conexas antes de executar o algoritmo.

Em relação a complexidade de espaço, o custo é guardar os elementos não explorados na fila e na fronteira, como são guardados só dois níveis da árvore na fronteira os espaço consumido pela fronteira é a combinação dos n voluntários i a i . A combinação de n elementos x a x é maior quando x é próximo de $\frac{n}{2}$, assim, no pior caso a complexidade de espaço é $O(\frac{n!}{((\frac{n}{2})!)^2})$. Outro gasto seria o de guardar o grafo, como usamos uma matriz de ad-

jacência, este custo é $O(n^2)$ que é bem menor que o custo de guardar a fronteira.

Para o caso médio temos que somar todos os casos ponderados por sua probabilidade de ocorrência. O evento considerado é o de ter uma solução de tamanho i , assim, a probabilidade desse evento é dada por, $\frac{\binom{n}{i}}{2^n}$ que é o número de combinações de n elementos i a i dividido pelo total de possibilidades. O custo de uma solução de tamanho i é dado por $\sum_{j=0}^i \binom{n}{j}(n + m + f + e_f)$. Dessa forma o custo médio é dado por $\sum_{i=0}^n \left(\frac{\binom{n}{i}}{2^n} \sum_{j=0}^i \binom{n}{j}(n + m + f + e_f) \right)$, aonde i representa o tamanho das possíveis soluções.

Case	Time	Space
Best	$\Theta(n(n + m + f + e_f))$	$\Theta(n^2 + n)$
Worst	$\Theta((2^n)(n + m + f + e_f))$	$O\left(\frac{n!}{((\frac{n}{2})!)^2} + f + e_f\right)$
Regular Case	$\sum_{j=0}^i \binom{n}{j}(n + m + f + e_f)$	$O\left(\frac{n!}{((\frac{n}{2})!)^2} + f + e_f\right)$

Tabela 1: Tabela com a complexidade de tempo e espaço para o melhor e pior caso do algoritmo considerando uma entrada de tamanho n . Para o caso geral temos uma fórmula em função da entrada e do tamanho da menor solução, ‘ i ’, dado pela terceira linha da tabela.

Como representado na Tabela 1, o caso geral é menor ou igual a 2^n em tempo, pois:

$$\sum_{j=0}^{i|i < n} \binom{n}{j}(n + m + f + e_f) \leq \sum_{j=0}^n \binom{n}{j}(n + m + f + e_f) = 2^n$$

Entretanto esse método tem uma complexidade de espaço muito grande.

3.2 Complexidade da Solução por Busca em Profundidade

A complexidade da solução por busca em profundidade pode ser vista na Tabela 2.

Sem realizar nenhuma poda o custo é igual para todos os casos, isto é toda a árvore é pesquisada e por isso gasta-se $\emptyset(n^f(n + e_v))$, que é o *branching factor*, n elevado a altura da árvore, que é f . Entretanto, se após tendo achado uma solução de tamanho ‘ j ’ não explorarmos nenhum nó cujo estado tem tamanho maior ou igual a ‘ j ’, já conseguimos alguma melhora, na prática, e o melhor caso passa a ter custo $\Theta(f(n + e_v))$, que é quando após explorar f nós, a profundidade inteira da árvore, achamos a melhor solução, e esta tem tamanho 1, assim, nenhum outro nó será investigado.

No pior caso, todas as folhas são exploradas e verificadas, assim, temos uma complexidade de $O(n^f(n + e_v))$, note que na prática o *branching factor*, base da exponencial, é normalmente menor que n , por isso usamos a notação O e não Θ .

Para o caso médio, consideramos o valor esperado do evento em que i nós folhas foram explorados. Também consideramos que c é o custo de explorar nós-não folhas. Dessa forma, o custo médio é dado por:

$$\begin{aligned}
& \sum_{i=1}^{n^f} \left(\frac{i}{n^f} (n + e_v) + c \right) \\
& \frac{1}{n^f} \sum_{i=1}^{n^f} (i(n + e_v) + c) \\
& \frac{1}{n^f} \left(\sum_{i=1}^{n^f} i(n + e_v) + \sum_{i=1}^{n^f} c \right) \\
& \frac{1}{n^f} \left((n + e_v) \sum_{i=1}^{n^f} i + \sum_{i=1}^{n^f} c \right) \\
& \frac{1}{n^f} \left((n + e_v) \left(\frac{(1+n^f)n^f}{2} \right) + c * n^f \right) \\
& (n + e_v) \left(\frac{(1+n^f)}{2} \right) + c
\end{aligned}$$

Como o termo c é o custo de explorar os nós não-folha, e estes são em menor quantidade que os nós folhas (v^f), então a complexidade do caso médio se torna: $O(n^f(n + e_v))$

Em relação a complexidade de espaço, como temos uma busca em profundidade, o gasto é no máximo da ordem da altura da árvore, que é $\Theta(f)$, mas, também temos o custo de guardar os dois grafos, que equivale a $\Theta(f + e_f + n^2)$, pois usamos uma matriz de adjacência pro grafo de amizades e uma lista de adjacência pro grafo de focos. Assim a complexidade de espaço final é $\Theta(2f + e_f + n^2) = \Theta(f + e_f + n^2)$.

Case	Time	Space
Best	$\Theta(f + n + e_v)$	$\Theta(f + e_f + n^2)$
Worst	$O(n^f(n + e_v))$	$\Theta(f + e_f + n^2)$
Average	$O(n^f(n + e_v))$	$\Theta(f + e_f + n^2)$

Tabela 2: Tabela com a complexidade de tempo e espaço para o melhor e pior caso do algoritmo considerando uma entrada com n voluntários, f focos, e_v amizades e e_f arestas entre voluntários e focos.

4 Exercício 5 - Experimentos

Nesta Seção relatamos os experimentos realizados para testar os algoritmos desenvolvidos para solucionar o problema.

Nas Figuras 1 e 2 reportamos os resultados do método de busca em largura, podemos ver que no caso de soluções de tamanho 1 (um) o tempo gasto é muito pequeno, entretanto, para o caso de soluções maiores a curva de tempo cresce mais rápido do que a exponencial, mostrada em azul. Já nas Figuras 4 e 3 os resultados da busca em profundidade são reportados, podemos perceber que tais resultados mostraram uma curva com crescimento bem menor que a exponencial. Por fim, as tabelas 4 e 3, mostram o tempo

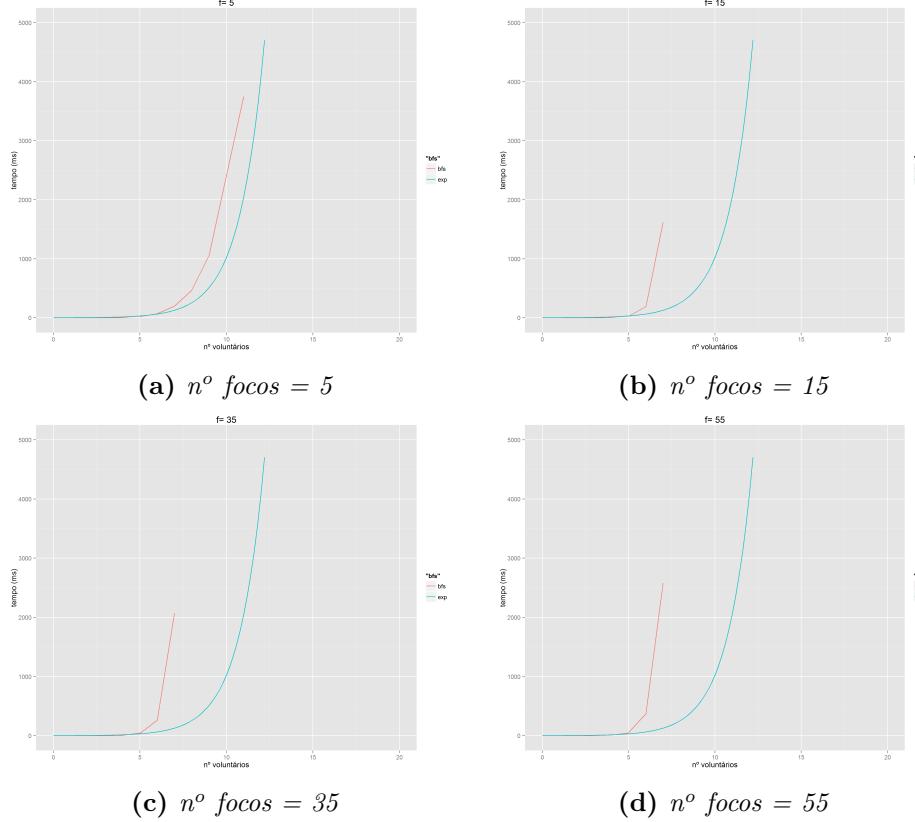


Figura 1: Execução do método por busca em largura para vários tamanhos de focos, e de voluntários. Nesses testes a menor solução era formada por todos os voluntários.

de execução de entradas grandes quando o tamanho da solução é 1(um), isto para demonstrar a eficiência da busca em largura nesses casos.

#Voluntários	Focos	Tempo(ms)
500	500	2290
1000	1000	8925
2000	2000	35515

Tabela 3: Relatório do tempo gasto pela solução de profundidade nos focos para valores de n grandes nos casos de soluções ótimas de tamanho 1 (um).

Como mostra de corretude, para rodar e comparar as saídas da implementação com as respostas disponibilizadas pelo monitor, foram gastos 21 milissegundos. Na Figura 5 mostramos o script de windows batch usado para rodar os testes.

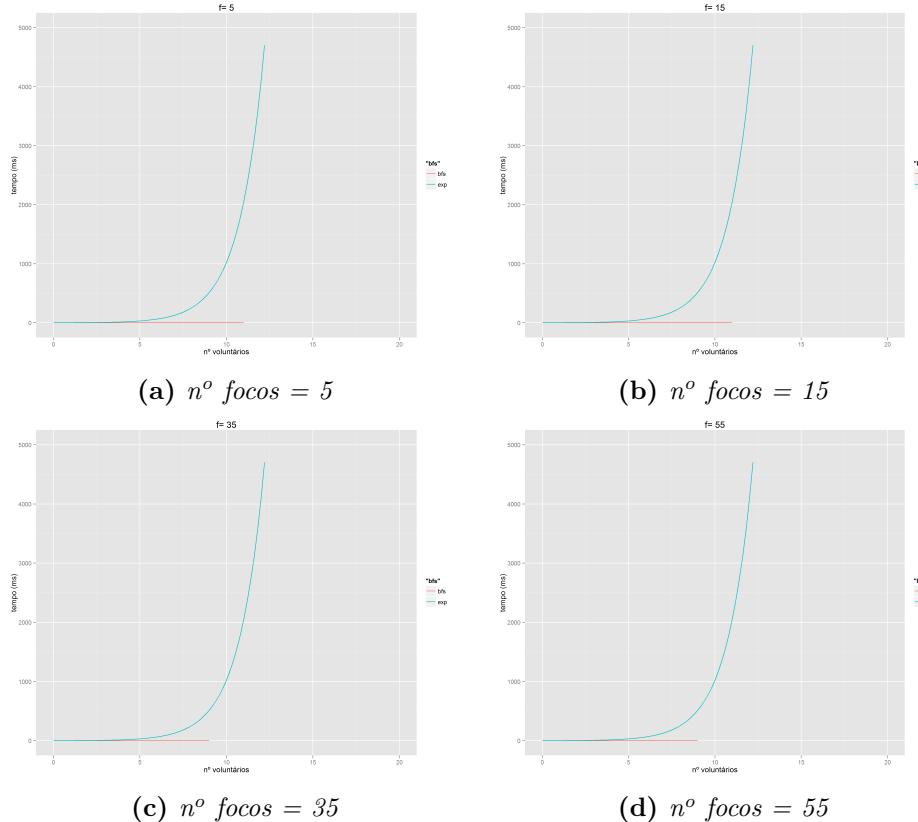


Figura 2: Execução do método por busca em largura para vários tamanhos de focos, e de voluntários. Nesses testes a menor solução era formada por um único voluntário.

#Voluntários	Focos	Tempo(ms)
500	500	390
1000	1000	1573
2000	2000	6143

Tabela 4: Nessa Tabela podemos, ver que apesar da solução por busca em largura ter sido menos eficiente em geral, a mesma obtém melhores resultados para valores de n grande em que a melhor solução tem tamanho pequeno, nesse caso, tamanho um.

5 Exercício 6 - Discussão

Nesta seção discutimos os resultados e suas consequências.

Na seção anterior 4 pudemos, ver que a busca em largura obteve um resultado na prática próximo do seu resultado teórico, isto é, ela se comportou com um crescimento maior que a curva exponencial, o que era esperado visto que sua complexidade de tempo foi obtida como $\Theta((2^n)(n + m + f + e_f))$

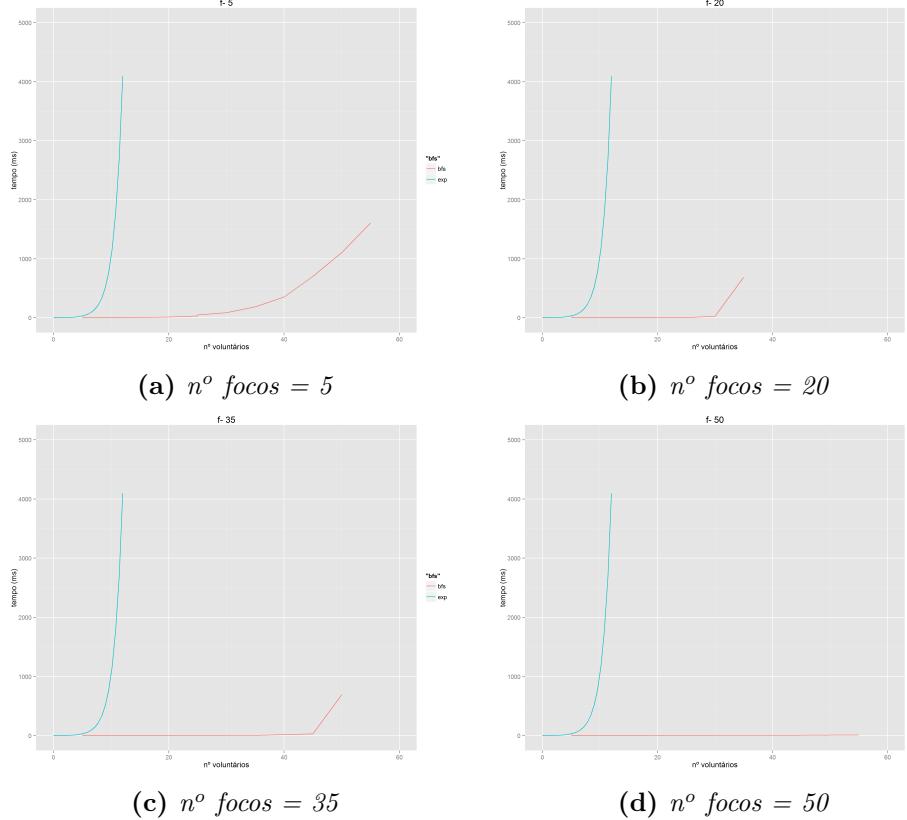


Figura 3: Execução do método por busca em profundidade para vários tamanhos de focos, e de voluntários. Nesses testes a menor solução era formada por todos os voluntários.

no pior caso. Já a complexidade para a busca em profundidade pelos focos obteve um desempenho melhor que o teórico. Sua complexidade foi calculada como $O(n^f(n + e_v))$ no caso médio e pior caso, entretanto teve um crescimento menor que o da curva exponencial. Entendemos que isso se dá porque a complexidade encontrada é um limite superior e na prática, fatores como o *branching factor* e a poda, o fato de não se procurar soluções de tamanho maior que as já encontradas, contribui muito para diminuir o custo do método, entretanto o *branching factor* real só pode ser calculado com um forte conhecimento das entradas do problema e sem esse conhecimento, é melhor assumí-lo como sendo n , que é o máximo que pode assumir. O impacto da poda foi considerado na modelagem do caso médio do método, entretanto, considerávamos que se tinha uma quantidade de testes muito grande e variados, nos casos de testes feitos usamos soluções de tamanho 1 e n , pela facilidade de modelá-las e reproduzí-las, esse fato pode ter enviesado as entradas dos testes, e assim, devido a esses fatores, o algoritmo acabou por apresentar um comportamento polinomial.

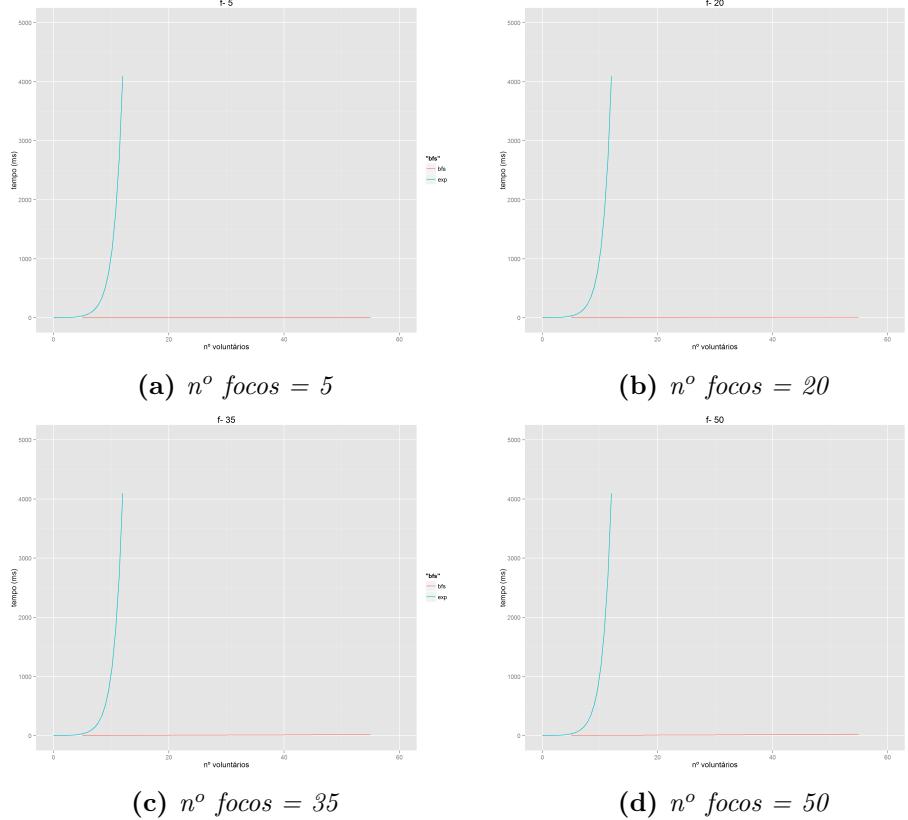


Figura 4: Execução do método por busca em profundidade para vários tamanhos de focos, e de voluntários. Nesses testes a menor solução era formada por um único voluntário.

Para a realização dos testes pelo monitor, o algoritmo escolhido foi a busca em profundidade, pois a busca em largura nos testes realizados começou a gastar uma quantidade de tempo muito grande em cerca de uma entrada de tamanho $n = 10$.

Como conclusões finais, tiramos como ponto mais importante do trabalho o fato que análises teóricas podem ter resultados bem diversos quando submetidos a prática, isso pode ocorrer devido a constantes altas que são ocultas na análise de complexidade, ou, no caso apresentado nesse trabalho, devido a propriedades do domínio em que se está aplicando o algoritmo.

```

@echo off
setlocal EnableDelayedExpansion

set inpName=input
for %%I in (in*.txt) do (
    set inpName=%%I
    ..\build\bin\Debug\PAA-zika.exe %%I >
        m_out!inpName:~2!
)
for %%I in (out*.txt) do (
    fc m%%I %%I
)

```

Figura 5: bat script para rodar os testes e verificar suas saídas.

Figura 6: Screenshot da execução dos testes disponibilizados. Foram gastos 21 ms para executar todos eles.

TP Grafos - ZikaZeroZ

Alex de Paula Barros¹

¹Universidade Federal de Minas Gerais
Belo Horizonte, MG, Brasil

{alexbarros}@dcc.ufmg.br

Exercício 1

Seja o grafo $G(\mathcal{V}, \mathcal{A})$ não direcionado, sendo \mathcal{V} o conjunto de n voluntários, sendo $n > 0$, e \mathcal{A} o conjunto de m relações de amizade, seja também \mathcal{F} o conjunto de focos de reprodução do mosquito, sendo $\mathcal{F} \neq \emptyset$, e a função $w(v) : \mathcal{V} \rightarrow \mathcal{F}$ definida para todo $v \in \mathcal{V}$, explicitando a quais focos o voluntário v tem acesso. Deseja-se obter o grafo $G'(\mathcal{V}', \mathcal{A}')$ tal que G' é conexo, $\mathcal{V}' \subseteq \mathcal{V}$, $\mathcal{A}' \subseteq \mathcal{A}$, $\bigcup_{v \in \mathcal{V}'} w(v) = \mathcal{F}$ e $|\mathcal{V}'|$ seja mínimo.

Exercício 2

Algorithm 1 Backtrack Solution

```
Visited =  $\emptyset$ 
BestSolution =  $\emptyset$ 
start = getStartingNode() //O(| $\mathcal{F}$ |)
visit(start)
```

Algorithm 2 visit(v)

```
Visited = Visited  $\cup$  {v}
if isSolution(Visited) //O(| $\mathcal{F}$ |) then
    BestSolution = better(BestSolution, Visited) //O(| $\mathcal{V}$ |)
else
    for adj in v.adjascents do
        if !isVisited(adj) // $\Theta(1)$  then
            visit(adj)
    parentVisit()
    Visited = Visited - {v}
```

Algorithm 3 parentVisit()

```
for adj in node.adjascents do
    if !isVisited(adj) then
        visit(adj)
```

O método `getStartingNode()` utiliza uma estrutura de dados para armazenar por quantos vértices um foco f é acessível. Se existir um foco acessível a partir de apenas um vértice este vértice garantidamente está na solução, caso contrário um vértice artificial ligado a todos os outros vértices é retornado. As arestas do vértice artificial são direcionadas.

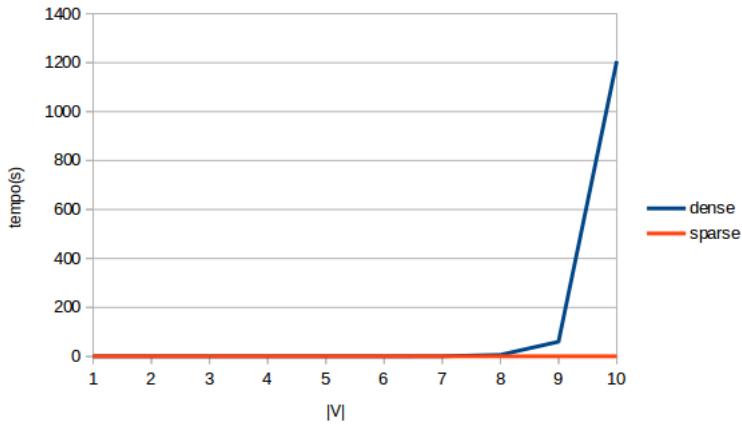


Figura 1. Tempo de execução do algoritmo em relação ao tamanho da entrada

Exercício 3

O método *getStartingNode*, descrito no exercício anterior, possui complexidade de tempo de $O(|\mathcal{F}|)$, já *visit* busca todas as soluções possíveis a partir de v até todos os outros vértices não visitados, no pior caso (grafo conexo), *visit* é chamado $\Theta(|\mathcal{V}|!)$, mesmo número de vezes que *parentVisit* é invocado. A cada vez que *visit* é chamado, *isVisited*, que executa um número constante de operações, é invocado $O(|\mathcal{V}|)$ vezes, e *isSolution* é invocado uma vez e executa $O(|\mathcal{F}|)$ operações. Assim a complexidade assintótica do algoritmo é $O(|\mathcal{V}|! * |\mathcal{F}|)$. No melhor caso o vértice encontrado por *getStartingNode* é solução para o problema e é o único vértice visitado, logo o este algoritmo é $\Omega(|\mathcal{F}|)$.

Quanto a complexidade de espaço, para armazenar o grafo utilizando lista de adjacência, é necessário armazenar o vértice, os focos acessíveis por este e sua lista de adjacência. Logo a complexidade de memória é $O(|\mathcal{V}| * |\mathcal{F}| + |\mathcal{A}|)$. Como as demais estruturas de dados utilizadas para armazenar os vértices já visitados ($O(|\mathcal{V}|)$) e os focos alcançáveis ($O(|\mathcal{F}|)$) têm complexidade assintótica menor que a de armazenamento do grafo, a complexidade de espaço do algoritmo é $O(|\mathcal{V}| * |\mathcal{F}| + |\mathcal{A}|)$.

Exercício 6

O algoritmo desenvolvido foi testado utilizando as instâncias fornecidas juntamente com a especificação para verificar sua corretude. Para a avaliação do tempo de execução do algoritmo foram geradas instâncias semelhantes de grafos conexos e grafos esparsos variando $|\mathcal{F}|$, $|\mathcal{A}|$ e $|\mathcal{V}|$. O gráfico da figura 1 expõem os tempos de execução com relação à $|\mathcal{V}|$ para grafos conexos (*dense*) e grafos esparsos (*sparse*). O tempo de execução em grafos densos "explode" para grafos com $n > 8$, saltando de 6s para 20min quando o n sobe de 9 para 10, comportamento esperado de algoritmos com complexidade assintótica factorial. O tempo de execução para matrizes esparsas testadas (que eram basicamente uma lista encadeada) o tempo foi praticamente constante para n de tamanho até 10. As instâncias utilizadas para avaliação de desempenho serão entregues juntamente com o código, nas pastas *dense* e *sparse*.