

Trabalho Prático PAA

O Problema ZicaZeroAnelDual

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 anel (aquele em que cada vértice conecta-se exatamente a outros dois) $G = (\mathbb{V}, \mathbb{A})$, em que \mathbb{V} é o conjunto de n voluntários e \mathbb{A} é o conjunto de $m = |\mathbb{V}|$ 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, sendo que $|R(V)| = 2 \forall v \in \mathbb{V}$. 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. Proponha algoritmos de complexidade assintótica polinomial com o tamanho da entrada (n, m, r) para resolver o problema ZicaZeroAnelDual usando os paradigmas busca por força bruta, programação dinâmica e algoritmo guloso.

Força bruta O algoritmo que utiliza a busca por força bruta pode ser visto em 1. Este algoritmo basicamente cria uma fila circular com os voluntários presentes no grafo de amizades. A partir desta fila o algoritmo escolhe cada voluntário como ponto de partida uma vez e testa as combinações a partir deste ponto de partida adicionando um por um os voluntários à direita até que se chegue a uma solução válida. As soluções encontradas são comparadas entre si, sendo a melhor retornada. Este algoritmo garante a solução ótima.

Programação dinâmica O algoritmo por programação dinâmica pode ser visto em 2. O algoritmo primeiramente cria uma lista circular com os voluntários presentes no grafo por ordem de amizades entre eles. Em seguida é criada uma tabela para armazenamento de resultados dos focos alcançados. O registro presente na linha i e coluna j indica o conjunto de focos que consigo alcançar a partir do j -ésimo voluntário na lista utilizando os i voluntários imediatamente à sua direita na lista. Inicialmente o algoritmo calcula os conjuntos das linhas de ordem 2 (2^k , onde k é incrementado a cada iteração). Quando uma linha tem um registro que alcança todos os focos temos assim definido um intervalo de procura definido pelas últimas duas linhas calculadas da tabela. Após este pré-cálculo é efetuada uma busca binária neste intervalo de linhas pelo registro que utiliza o menor número de voluntários e alcança todos os focos. Os novos registros que vão sendo calculados ao longo do algoritmo utilizam a união de registros anteriores para serem formados. Este algoritmo garante o resultado ótimo.

Guloso O algoritmo guloso pode ser visto em 3. No início do algoritmo cada vértice recebe um coeficiente numérico que indica quantos focos distintos eu consigo alcançar a partir dele e dos seus vizinhos à direita e esquerda. O vértice com maior

coeficiente é adicionado na resposta. Posteriormente os vértices à direita e à esquerda da lista de vértices adicionados à solução atual são verificados para ver quais deles adicionam mais focos à solução que ainda não foram alcançados utilizando seu vizinho mais próximo que ainda não está na solução. Aquele com maior coeficiente é adicionado à solução. Este processo se repete até que se chegue em uma solução válida. Este algoritmo não garante a solução ótima.

Algorithm 1: Algoritmo de busca por força bruta

Data: $G = (\mathbb{V}, \mathbb{A})$, grafo que representa voluntários e laços de amizade.
Data: F , dicionário com a lista de focos conhecidos por cada voluntário.

- 1 . **Data:** r , o número total de focos
- 2 . **Result:** S , lista com os voluntários que devem ser escolhidos para alcançar os focos.
- 3 **begin**
- 4 $ring \leftarrow createRingList(G)$ /* Cria uma lista com a ordenação dos voluntários no grafo anel */
- 5 $best \leftarrow G.V$
- 6 **foreach** $ini \in ring$ **do**
- 7 $fvisited \leftarrow F[ini]$
- 8 $solution \leftarrow \emptyset$
- 9 $solution.append(ini)$
- 10 $explore \leftarrow ring.after(ini) \cup ring.before(ini)$
- 11 **while** $fvisited.length < r$ **do**
- 12 $solution.append(explorer.popFirst())$
- 13 $fvisited \leftarrow fvisited \cup F[solution.last()]$
- 14 **if** $solution.length < best.length$ **then**
- 15 $best \leftarrow solution$
- 16 **return** $best$

2. Analise as complexidades Temporais e Espaciais (usando notação Assintótica) dos algoritmos proposto no exercício 1.

Força Bruta Este algoritmo tem complexidade temporal $O(n^2)$, pois para cada voluntário presente no problema são feitos n testes de combinações de escolhas. Ele possui complexidade espacial $O(n + r)$, pois é utilizada uma lista para guardar os focos que já foram alcançados na solução sendo avaliada e uma lista com a solução atual.

Programação Dinâmica Este algoritmo tem complexidade temporal $O(n \log \frac{n^2}{2})$. Pois o cálculo de cada linha da tabela tem complexidade $O(n)$, e, na etapa de definição do intervalo de busca binária são avaliadas no máximo $\log(n)$ linhas, somado com a busca binária que vai avaliar no máximo $\log \frac{n^2}{2}$ (visto que o maior intervalo que pode ser encontrado na primeira etapa tem $\frac{n^2}{2}$ linhas). Ao multiplicar esta quantidade máxima de linhas que são calculadas nas duas etapas com o custo $O(n)$ de cada linha chegamos à equação citada. Ele possui complexidade espacial também $O(n \log \frac{n^2}{2})$ que define a quantidade máxima de registros que são armazenados.

Algorithm 2: Algoritmo de programação dinâmica

Data: $G = (\mathbb{V}, \mathbb{A})$, grafo que representa voluntários e laços de amizade.
Data: F , dicionário com a lista de focos conhecidos por cada voluntário.

1 . **Data:** r , o número total de focos
Result: S , lista com os voluntários que devem ser escolhidos para alcançar os focos.

2 **begin**

3 $ring \leftarrow createRingList(G)$ /* Cria uma lista com a ordenação dos voluntários no grafo anel */

4 $table \leftarrow createEmptyTable()$

5 $table.computeRow(1)$

6 **if** $table.rowContainsSolution(1)$ **then**

7 **return** $table.solutionInRow(1)$

8 $maximum \leftarrow 1$

9 $minimum \leftarrow 1$

10 **while** $not table.rowContainsSolution(maximum)$ **do**

11 $minimum \leftarrow maximum$

12 $maximum \leftarrow maximum * 2$

13 $table.computeRow(maximum)$

14 **return** $table.searchSolutionInRowRange(minimum, maximum)$

Algorithm 3: Algoritmo guloso

Data: $G = (\mathbb{V}, \mathbb{A})$, grafo que representa voluntários e laços de amizade.
Data: F , dicionário com a lista de focos conhecidos por cada voluntário.

1 . **Data:** r , o número total de focos
Result: S , lista com os voluntários que devem ser escolhidos para alcançar os focos.

2 **begin**

3 $solution \leftarrow \emptyset$

4 $volunteer \leftarrow (maxUnknownFocuses(G))$

5 $fvisited \leftarrow F[volunteer]$

6 $solution.append(volunteer)$

7 **while** $fvisited.length < r$ **do**

8 $volunteer \leftarrow$

9 $maxUnknownFocuses(solution.right(), solution.left())$

10 $fvisited \leftarrow fvisited \cup F[volunteer]$

11 $solution.append(volunteer)$

11 **return** $solution$

Guloso Este algoritmo tem complexidade $O(n)$ pois só pode escolher cada voluntário uma vez. E possui complexidade espacial $O(n + r)$, onde n é o tamanho máximo da resposta que ele armazena durante a execução e r é o tamanho máximo da lista de focos já conhecidos que é utilizada durante a execução.

3. Implemente os algoritmos proposto no exercício 2 na linguagem de programação C, C++, JAVA ou PYTHON.

A implementação em PYTHON segue em anexo ao relatório.

4. Execute testes da implementação do Exercício 3, propondo instâncias interessantes para o problema ZicaZeroAnelDual. Uma sugestão é que seja produzido um gráfico do tempo de execução por tamanho de entrada (n, m, r), comparando os três algoritmos implementados.

Foram executados testes com os exemplos disponibilizados no fórum da disciplina. Os seguintes resultados foram encontrados:

Algoritmo	Arquivo	n	m	r	Operações	Tempo(s)
Força bruta	in0	7	7	6	32	0.0032
P. Dinâmica	in0	7	7	6	21	0.0059
Guloso	in0	7	7	6	2	0.0135
Força bruta	in1	6	6	4	17	0.0018
P. Dinâmica	in1	6	6	4	6	0.0016
Guloso	in1	6	6	4	1	0.0013
Força bruta	in2	9	9	8	68	0.0024
P. Dinâmica	in2	9	9	8	45	0.0067
Guloso	in2	9	9	8	4	0.0017
Força bruta	in3	9	9	8	64	0.0024
P. Dinâmica	in3	9	9	8	27	0.0035
Guloso	in3	9	9	8	7	0.0017
Força bruta	in4	9	9	8	58	0.0038
P. Dinâmica	in4	9	9	8	27	0.0041
Guloso	in4	9	9	8	4	0.0028
Força bruta	in5	17	17	9	124	0.0048
P. Dinâmica	in5	17	17	9	85	0.0068
Guloso	in5	17	17	9	5	0.0022

Tabela 1. Tabela com testes a partir dos exemplos postados no fórum da disciplina

5. Compare a análise e a execução respectivamente, exercícios 2 e 4. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

Ao comparar os resultados dos testes com a previsão teórica a quantidade de operações importantes feitas em cada um dos algoritmos segue a previsão estimada no seu cálculo de

complexidade. Podemos ver também que apesar do algoritmo de programação dinâmica ter um limite superior de operações menor que o força bruta, ele alcança um tempo de execução equivalente ou até superior em alguns casos quando comparado com ele. Isso se deve ao fato do algoritmo de programação dinâmica efetuar operações de união entre conjuntos, que é uma operação mais complexa. Portanto para instâncias menores vale mais a pena usar o algoritmo de força bruta.

Documentação do Trabalho Prático ZikaZeroAnelDual

Alexandra da Silva Pereira¹

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

alesilva241@gmail.com

1. Proponha algoritmos de Complexidade Assintótica Polinomial com o Tamanho da entrada (n, m e r) para resolver o Problema ZikaZeroAnelDual usando os seguintes paradigmas:

1.1. Busca Por Força-Bruta

A estratégia utilizada para apresentar uma solução de força bruta que execute em tempo polinomial é bem simples. O procedimento de DFS (*Depth-first search*) é utilizado para percorrer todos os vértices em busca de uma lista conexa de vértices, começando sempre do vértice 1. Logo em seguida é executado um procedimento que trabalha sob a "lista conexa" de vértices de maneira similar a uma lista circular, onde cada novo círculo é formado a cada 1 salto. Esse procedimento de rotação é repetido de acordo com o número de vértices existente no grafo. A cada passada é obtida uma possível solução, que é comparada com possíveis anteriores soluções e caso a nova cubra todo o conjunto com menor número de vértices, então a solução ótima é atualizada. Como exemplo, no grafo traduzido pela instância *in0*, podemos exemplificar da seguinte forma:

- Lista retornada da DFS:
1, 4, 6, 5, 2, 7, 3

- Após uma rotação:
4, 6, 5, 2, 7, 3, 1
...

E o procedimento segue até que sejam dadas as n voltas, isto é, até que a busca por soluções seja iniciada a partir de cada um dos vértices em um mesmo sentido, no caso adotado na solução, em sentido anti-horário. Por fim apresentamos para essa solução o pseudo-código equivalente ao algoritmo implementado em linguagem *Python*, no bloco referente ao algoritmo 1: Procedimento Força-Bruta. Neste bloco é possível notar que o programa responsável por essa solução trabalha com 2 procedimentos. O primeiro deles é responsável pela DFS, que garante que todas as ordens exploradas na lista posteriormente sejam conexas. O segundo é a etapa referente a "rotação" dessa lista, em cada passada uma nova solução é formada, e caso seja uma solução menor (com menos vértices) haverá um *update* na possível solução já encontrada, mas que possui maior número de vértices.

```

1 Algoritmo brute-Force()
2 resolved  $\leftarrow \emptyset$ 
3 resolved = call rotateSolution(dfs_bruteForce(0, F), V, F)
4 Procedimento dfs_bruteForce(V, F, visitedVertex)
5 visitedVertex  $\leftarrow V$ 
6 for x  $\in V_{adjacentList}$  do
7   | if x is not in visitedVertex then
8     |   | dfs_bruteForce(V adjacentList Element, F, visitedVertex)
9   | end
10 end
11 Procedimento rotateSolution(listReturnedbyDFS, V, F)
12 solution  $\leftarrow \emptyset$ 
13 aux_solution  $\leftarrow \emptyset$ 
14 possibleSolution  $\leftarrow \emptyset$ 
15 circularList = listReturnedbyDFS
16 foreach x  $\in V$  do
17   | for y  $\in circularList$  do
18     |   | if length(possibleSolution)  $\neq F then
19       |     |   possibleSolution = possibleSolution  $\cup$  focusKnown by y
20       |     |   aux_solution = vertex y
21     |   | end
22     |   | if length(possibleSolution) == F then
23       |     |   | if length(solution) == 0 then
24         |       |     |   solution = aux_solution
25       |     |   | end
26       |     |   | if length(solution) > lengthaux_solution then
27         |       |     |   solution = aux_solution
28       |     |   | end
29     |   | end
30     |   | circularList() update 1 to right
31   | end
32 end
33 return solution$ 
```

Algoritmo 1: Procedimento Força-Bruta

1.2. Programação Dinâmica

Para a estratégia de programação dinâmica pensou-se em uma maneira de armazenar possíveis soluções e evitar re-cálculo de operações previamente resolvidas. Começamos a exploração da árvore apresentada na figura 1 pelos nós folhas, a partir deles buscamos subconjuntos de vértices que possuem uma possível solução, seja ela ótima ou não. Observe os nós grifados em preto, eles representam subconjuntos que não apresentam solução. Os nós grifados em azul buscam demonstrar a sobreposição de operações, isto é, cada um desses subconjunto passa por uma operação que faz a união de focos cobertos por cada vértice apenas uma vez. Os nós grifados em vermelho apresentam um subconjunto solução que é considerado em conjuntos que o possuem como subconjunto.

Como a implementação dessa estrutura de "memória" da programação dinâmica foi resolvida em programação com um dicionário *Python*, no qual as *keys* são representadas pelo conjunto, e o *value* pela solução fornecida pelo conjunto que forma uma solução, de maneira mais clara teremos:

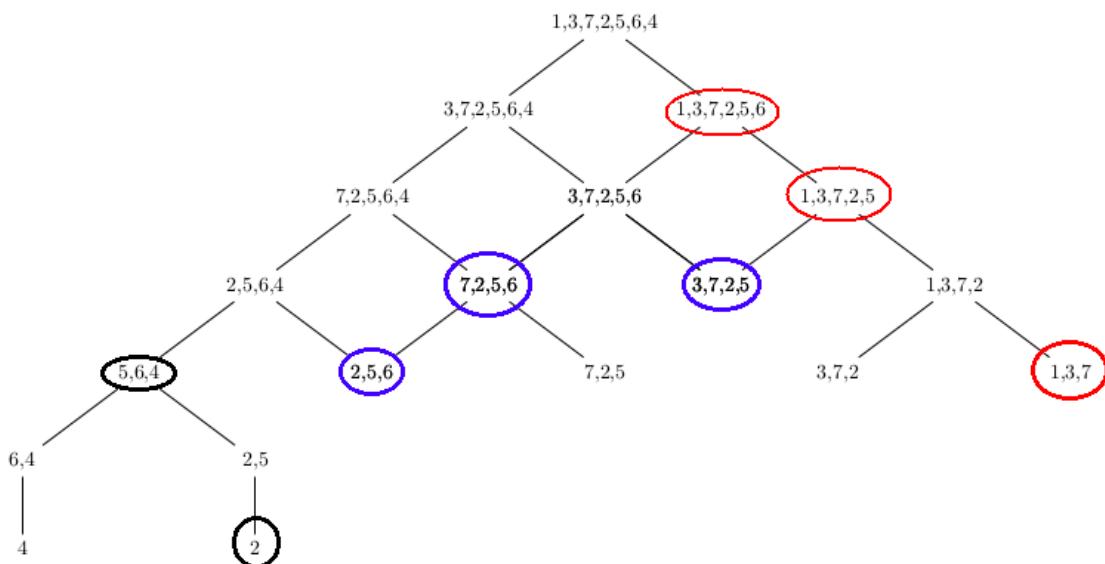
$$\begin{aligned}
 & V2 : \text{None} \\
 & V2, V5 : \text{None} \\
 & V4, V5, V6 : \text{None} \\
 & \dots \\
 & V1, V3, V7 : V1, V3, V7 \\
 & V1, V3, V7, V2, V5 : V1, V3, V7
 \end{aligned}$$


Figura 1. Exemplo de árvore para a instância *in0* na solução de PD.

```

1 Algoritmo dynamicGraph()
2   resolved  $\leftarrow \emptyset$ 
3   resolved = call start_pd( $V, F$ )
4   Procedimento start_pd()
5     vertex  $\leftarrow \emptyset$ 
6     result  $\leftarrow \emptyset$ 
7     for  $x \in V$  do
8       | call dfs_start( $x$ ) vertex  $\leftarrow$  inicialSolution
9       | if  $x == 0$  then
10      |   | result  $\leftarrow$  dp_solution( $F, vertexList$ )
11      | end
12      | if length(dp_solution( $F, vertexList$ ))  $\leq$  length(result) then
13      |   | result  $\leftarrow$  dp_solution( $F, vertexList$ )
14      | end
15      | inicialSol  $\leftarrow \emptyset$ 
16      | vertexList  $\leftarrow \emptyset$ 
17      | return result
18   end
19   Procedimento dfs_start( $V$ )
20   for adjacent vertex to  $V$  do
21     | if adjacent vertex to  $x$  is not in inicialSolution then
22     |   | inicialSolution  $\leftarrow$  adjacent vertex
23     |   | call dfs_start(adjacent vertex)
24     | end
25   end
26   Procedimento dp_solution()
27   aux_set  $\leftarrow \emptyset$ 
28   if length(vertex) == 0 then
29   | return Null
30   end
31   if vertexList  $\in$  memoryDP then
32   | return memoryDP where is vertexList
33   end
34   s1  $\leftarrow$  dp_solution( $F, \text{trunked } 1 \text{ vertexList}$ )
35   s2  $\leftarrow$  dp_solution( $F, \text{trunked } -1 \text{ vertexList}$ )
36   if s1 == Null and s2 == Null then
37     | foreach vertex from vertexList do
38     |   | aux_set  $\leftarrow$  aux_set  $\cup$  focusKnown by vertex
39     |   | if length(aux_set) ==  $F$  then
40     |   |   | memoryDP[vertexList]  $\leftarrow$  vertexList
41     |   |   | else
42     |   |   |   | memoryDP[vertexList]  $\leftarrow$  Null
43     |   | end
44     | end
45   end
46   end
47   if s1 == Null and not s2 == Null then
48   | memoryDP[vertexList]  $\leftarrow$  s2
49   end
50   if s2 == Null and not s1 == Null then
51   | memoryDP[vertexList]  $\leftarrow$  s1
52   end
53   if not s1 == Null and not s2 == Null then
54   | if length(s1) > length(s2) then
55   |   | memoryDP[vertexList]  $\leftarrow$  s2
56   |   | else
57   |   |   | memoryDP[vertexList]  $\leftarrow$  s1
58   |   | end
59   | end
60   end
61 return memoryDP[vertexList]

```

Algoritmo 2: Procedimento Dinâmico

1.3. Algoritmo Guloso

Neste trabalho a estratégia gulosa proposta para o problema surge de uma modelagem com a ideia de inclusão de pesos nas arestas. Os pesos representam o tamanho do conjunto união de focos dos vértices que são conectados por uma determinada aresta. Por exemplo na imagem 2 essa ideia é demonstrada de maneira clara, no qual, a união dos vértices V_1 e V_3 foram um conjunto composto de 4 focos, logo o peso da aresta que conecta tais vértices deve ser 4.

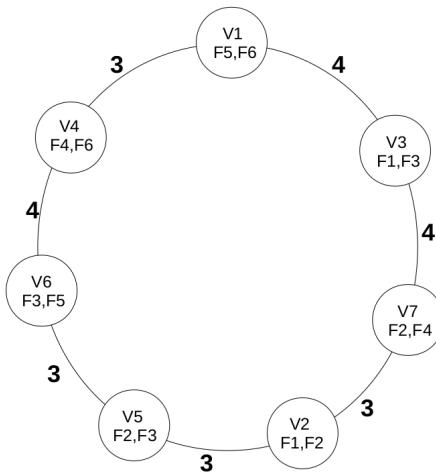


Figura 2. Representação da instância $in0$ para modelagem com arestas .

No exemplo apresentado a seguir temos uma demonstração da formação desse conjunto de 4 elementos.

$$\begin{aligned} & \text{Focos cobertos por } V_1 \cup \text{Focos cobertos por } V_3 \\ & \{F_5, F_6\} \cup \{F_1, F_3\} = \{F_5, F_6, F_1, F_3\} \end{aligned}$$

Os pesos das arestas começam a ser utilizados no algoritmo na etapa inicial, isto é, na **ordenação**. As arestas mais interessantes terão peso menor, isso porque adotou-se na implementação o inverso do valor que representa os pesos, no qual 4 é na verdade $1/4$ (0.25), e 0.25 é o peso da aresta. Isso fará com que no início da lista estejam a(as) aresta(as) que conecta(m) vértices com a maior cobertura de focos. Nessa lista ordenada sempre pegamos a primeira aresta e incluímos seus dois vértices na lista de solução, essa é a escolha gulosa. Na segunda etapa resolvemos um subproblema: Qual das arestas adjacentes será a adiciona a solução? E a solução segue a mesma estratégia gulosa, sempre busca adicionar mais focos a solução e a escolha sempre é aresta mais leve, caso não exista empate. No caso de empate sempre o primeiro valor visitado é levado a solução.

No algoritmo 3 apresentamos um pseudo-código do procedimento guloso proposto, e que foi implementado. Vale a ressalva de que essa estratégia é aproximada e não obteve os resultados ótimos de cada instância.

```

1 Algoritmo greedyGraph()
2   resolved  $\leftarrow \emptyset$ 
3   resolved = call adjMatrix(V, F)
4 Procedimento adjMatrix()
5   vertex_sol  $\leftarrow \emptyset$ 
6   for  $x \in V$  do
7     for  $y \in \text{adjacent vertex to } x$  do
8       weight  $\leftarrow \text{length}(\text{focusKnown by } x \cup \text{focusKnown by } y)$ 
9       adjacentMatrix  $\leftarrow \text{sorted}(\text{adjacentMatrix}, \text{by edge weight})$ 
10    end
11 end
12 call greedy_solution(F, vertex_sol)
13 return vertex_sol
14 Procedimento greedy_solution()
15   vertex_sol  $\leftarrow \text{get the vertex of first edge}$ 
16   call specialChoice_greedy(F, vertex_sol)
17 Procedimento specialChoice_greedy()
18   x_testing  $\leftarrow 0$ 
19   x_choiced  $\leftarrow 0$ 
20   focusCover  $\leftarrow \emptyset$ 
21   for  $x$  to  $\text{length}(\text{vertex\_sol})$  do
22     | focusCover  $\leftarrow \text{focusCover} \cup \text{focusKnown by vertex\_sol}[x]$ 
23   end
24   for  $x$  to  $\text{length}(\text{vertex\_sol})$  do
25     for  $y \in \text{vertex\_sol}[x]$  do
26       if  $y$  is not in vertex_sol then
27         if  $x\_testing < \text{length}(\text{focusCover} \cup \text{focusKnown by } y)$ 
28           |  $x\_testing = \text{length}(\text{focusCover} \cup \text{focusKnown by } y)$ 
29           | x_choiced = vertex_choiced
30         end
31       end
32     end
33   end
34   vertex_sol  $\leftarrow x\_choiced$ 
35   focusCover  $\leftarrow \text{focusCover} \cup \text{focusKnown by } x\_choiced$ 

```

Algoritmo 3: Procedimento Guloso

2. Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) dos algoritmos propostos no Exercício 1.

Complexidades Temporais

1. Força-Bruta

- **Melhor caso:** Na complexidade de tempo, de acordo com o algoritmo proposto, o melhor caso é quando a lista retornada pela DFS é a solução ótima, entretanto todas as rotações são feitas para que garantidamente seja

essa a melhor solução, isso tudo é feito em tempo polinomial, levando a complexidade de tempo a $O(v^2 + e) = O(v^2)$.

- **Pior Caso:** Na complexidade de tempo, de acordo com o algoritmo proposto, o pior caso é quando todas as rotasções são necessárias, todo o percurso tem custo $O(v^2 + e) = O(v^2)$.

2. Programação Dinâmica

- **Melhor caso:** Ocorre quando a folha da árvore é a solução ótima e as operações de conjuntos não precisam ser feitas novamente, uma vez que já estará na tabela, o que ocasiona apenas em uma consulta na tabela, pois uma chave é subconjunto da chave atual e já está calculada sua solução. A cópia de um valor, para associação com uma chave que possui em seus elementos um subconjunto já calculado como solução ótima, consiste em complexidade $O(v)$.
- **Pior Caso:** Esse caso já é o inverso do melhor caso, no qual o conjunto formado apenas por todos os vértices é solução. Isso faz com que a busca e a construção pela solução seja $O(v^2 * v)$, que pode ser generalizado como: $O(v^3)$.

3. Gulosos

- **Melhor caso:** No melhor caso do algoritmo guloso fazemos a operação responsável pela ordenação de todo o conjunto de arestas, isso leva a complexidade de tempo igual a $O(v)$, entretanto ao executar a operação de escolhas, podemos fazer apenas 1 escolha, ou seja $O(1)$, pois um vértice cobre todo o conjunto de focos. Com isso teremos complexidade temporal de $O(n) + O(1)$ que pode ser generalizado para $O(v)$.
- **Pior Caso:** No pior caso o algoritmo fará a ordenação do conjunto de arestas, explorando assim todo o conjunto v de vértices, que é o mesmo ocorrido no melhor caso, com complexidade de tempo igual a $O(v)$. Mas na segunda etapa do algoritmo guloso, ele também necessitará percorrer todo o conjunto de vértices, partindo do pressuposto que apenas todo o conjunto formará um conjunto solução, logo novamente $O(v)$ que pode ser generalizado da seguinte maneira: $O(v + v) = O(2v)$, ou ainda apenas como: $O(v)$.

Complexidades Espaciais

1. Força-Bruta

No força-bruta a complexidade espacial está fortemente concentrada no armazenamento do grafo e no array de solução, com respectivamente $O(v + e)$ e no máximo $O(v)$ quando todos os vértices fazem parte da solução. Ao longo do programa são criadas estruturas auxiliares que armazenam uma quantidade de valores no máximo igual a quantidade de vértices, ou seja, $O(v)$. Levando a uma complexidade espacial igual $constante * v$ que pode ser generalizado como $O(v)$.

2. Programação Dinâmica

A complexidade espacial necessária para essa solução se concentra principalmente na construção de um dicionário *Python* que armazena um subconjunto e se ele possui solução ou não, isso implicará em $O(2^n)$ de espaço necessário, que mesmo exponencial não possuem relação direta com complexidade temporal e

representa o consumo de bem poucos bytes. Ao longo da solução são utilizadas variáveis auxiliares, mas que não impactam fortemente na solução, pois são valores que estão entre 1 e $O(n)$, arrays referentes ao grafo ocupando espaço $O(n + e)$, e ao número total de vértices, com complexidade no máximo $O(n)$. Portanto, $O(2^n) + O(n + e) + O(n) + O(1) = O(2^n)$.

3. Gulosos

Na estratégia gulosa a complexidade espacial está fortemente concentrada no armazenamento do grafo e no array de solução, com respectivamente $O(v + e)$ e no máximo $O(v)$ quando todos os vértices fazem parte da solução. Ao longo do programa são criadas estruturas auxiliares que armazenam uma quantidade de valores no máximo igual a quantidade de vértices, ou seja, $O(v)$. Levando a uma complexidade espacial igual a $constante * v$ que pode ser generalizado como $O(v)$.

3. Implemente os algoritmos propostos no Exercício 1 na linguagem de programação C, C++, Java ou Python.

Os algoritmos foram implementados em linguagem *Python* e o código de cada um deles pode ser encontrado no diretórios de cada projeto dentro da pasta implementações.

4. Execute testes da implementação do Exercício 3, propondo instâncias interessantes para o Problema ZikaZeroAnelDual. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n, m e r) comparando os três algoritmos implementados.

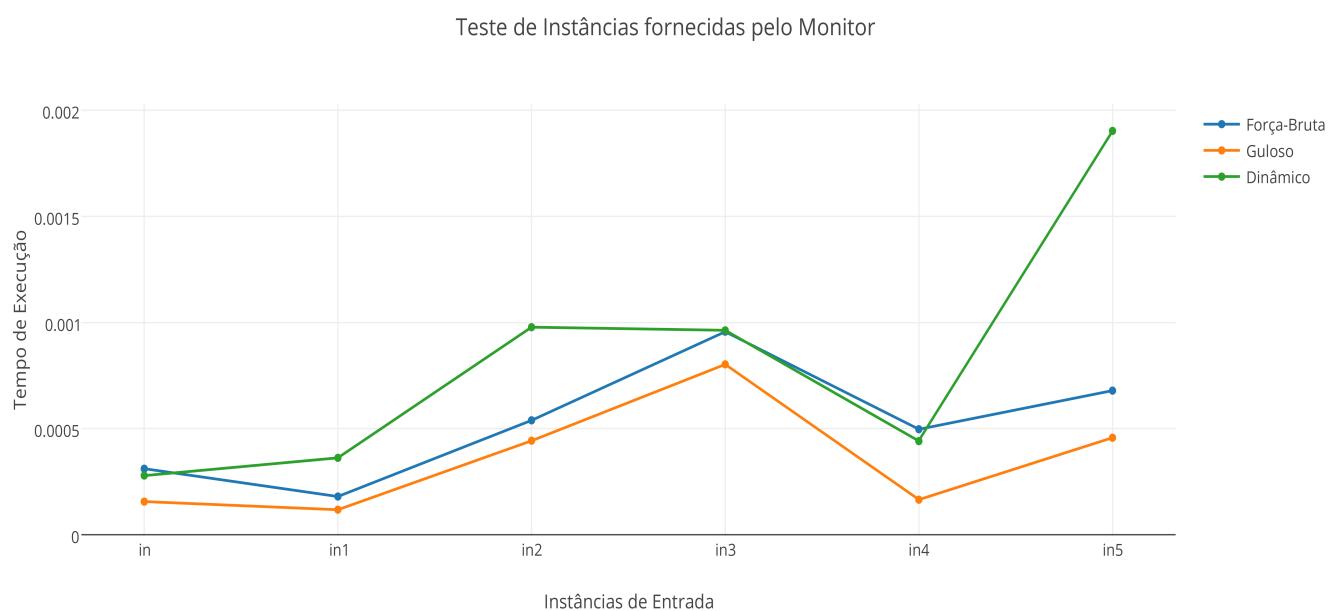


Figura 3. Gráfico de Instâncias Fornecidas pelo Tempo de Execução.

Teste Realizado com Instâncias Maiores (Tam. Entrada x Temp Execução)

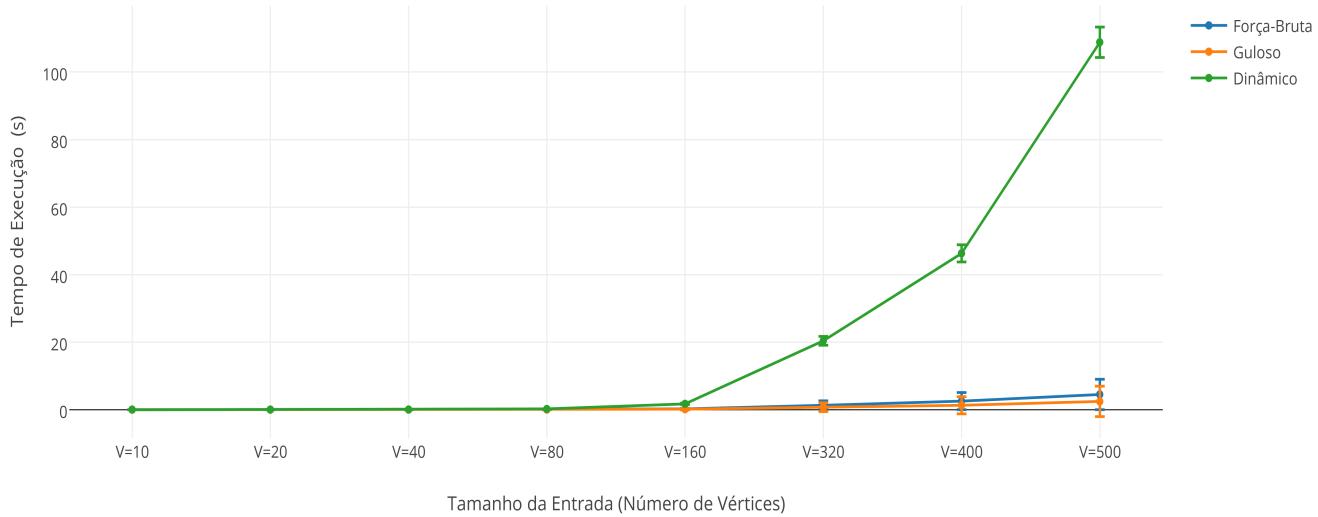


Figura 4. Gráfico de Instâncias Maiores pelo Tempo de Execução.

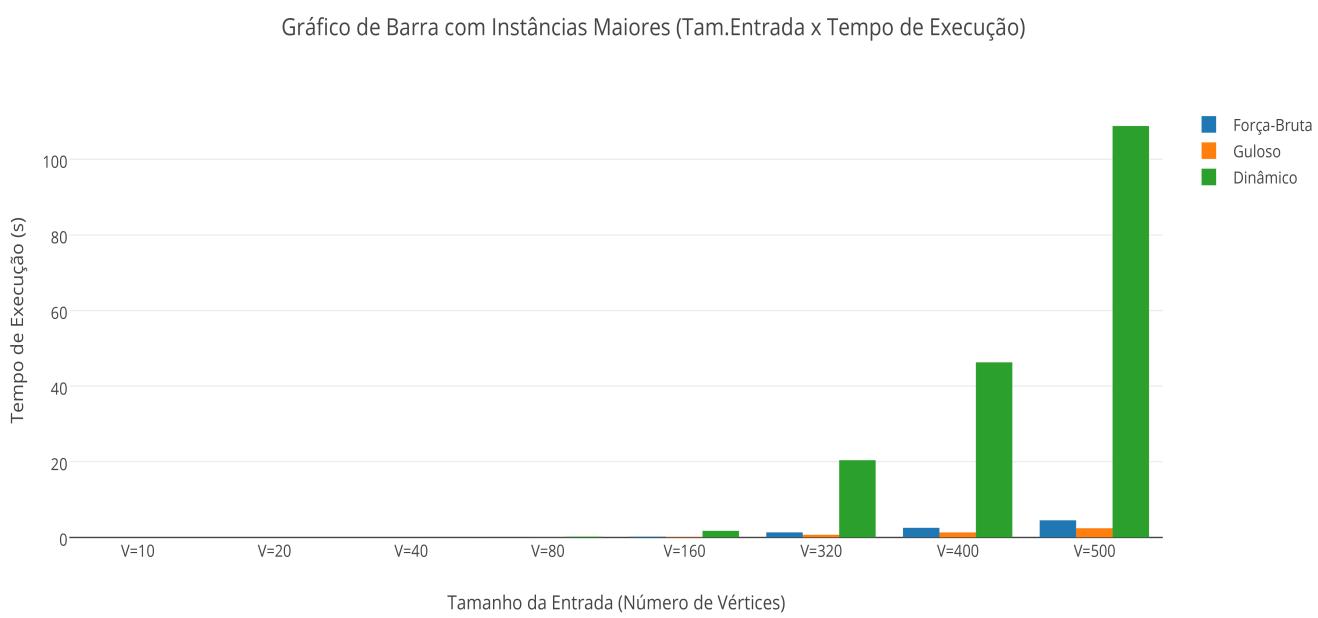


Figura 5. Gráfico de Barra das Instâncias Maiores pelo Tempo de Execução.

**5. Compare a análise e a execução, respectivamente, Exercícios 2 e 4.
Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.**

Como esperado, as previsões teóricas foram confirmadas com os resultados experimentais, com isso, as previsões teóricas estão em concordância com as proposições teóricas discutidas na questão 2. Os gráficos 4 e 5 tornam ainda mais clara a análise apresentada teoricamente e sua complexidade de tempo real.

PAA - Trabalho Prático 2: ZikaZeroAnelDual

Thiago Carvalho D'Ávila¹

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

thiagoc@ufmg.br

1. Propostas de algoritmos polinomiais (Exercício 1)

Dados um conjunto U de elementos (que no caso são os r focos) e uma coleção de subconjuntos $S = \{S_1, S_2, \dots, S_k\}$ (que seriam os focos que cada voluntário cobre), o problema em se encontrar o menor número de subconjuntos tal que a união dos mesmos cubra todo o conjunto U é conhecido na literatura como *Set-Cover* [Vazirani 2001].

Há uma extensão desse problema chamada (*Minimum*) *Connected Set-Cover* (*MCSC*) definido a partir de um grafo $G = (S, E)$, em que S são subconjuntos de U e E o conjunto de relações entre um par de voluntários de S .

Tanto o *Set-Cover* quanto o *MCSC* são problemas NP-Completo, entretanto o ZikaZeroAnelDual trata-se de uma instância específica do *MCSC* em um grafo em anel, o que torna possível o uso de algoritmos eficientes / polinomiais [Shuai and Hu 2006]. Isso ocorre de maneira análoga em outros conhecidos problemas NP-Completo como a instância 2-SAT do problema *SAT* e a instância 2-coloring do problema *k-coloring* (coloração de grafos).

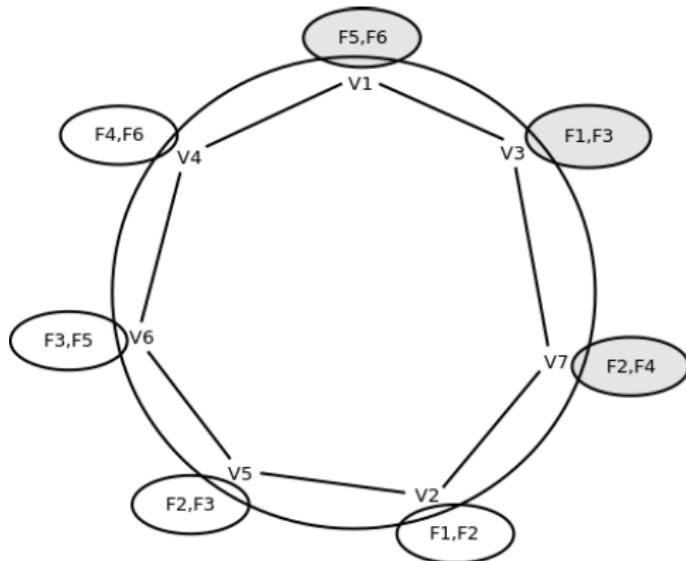


Figura 1. MCSC em grafo anel

Devido à restrição do grafo anel de que cada vértice de subconjuntos tem grau dois (Figura 1), o grafo G se torna esparsa e pode representar suas arestas eficientemente como uma lista de adjacências, reduzindo o custo de espaço de uma matriz de adjacências $|S^2|$ para um custo linear $|2S|$.

Desta forma, os três algoritmos propostos irão ler o arquivo de entrada e preencher a lista de adjacências RELATIONSHIP_DICT, gerarão o conjunto universo U com os r focos UNIVERSE_SET e preencherão a lista de conjuntos com os focos cobertos por cada voluntário VOLUNTEER_COVER .

1.1. Força-bruta (FB)

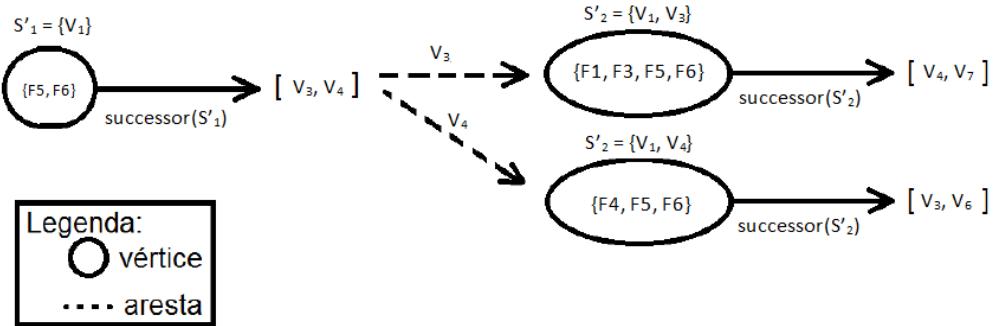


Figura 2. Função sucessor()

Seja S' a coleção de subconjuntos de S atual, que possui uma cobertura U' de U . Define-se em $G = (S, E)$ uma função **sucessor** (Figura 2) de S' que retorna os conjuntos conectados adjacentes a S' presentes em E . No caso específico do ZikaZeroAnelDual, o retorno terá grau máximo dois, que são as possíveis transições a serem exploradas pelo algoritmo força-bruta (e também pelo de programação dinâmica).

Algoritmo 1: Algoritmo ZikaZeroAnelDual força-bruta

Result: $\text{melhor_solucao} \vee \text{NIL}$

```

1  $\text{melhor\_solucao} \leftarrow \text{NIL};$ 
2  $\text{soma\_melhor\_solucao} \leftarrow 0;$ 
3 foreach  $\text{subconjunto} \in S$  do
4    $\text{novo\_vertice} \leftarrow \text{subconjunto};$ 
5    $\text{nova\_solucao} \leftarrow \text{BFS}(\text{novo\_vertice});$ 
6   if  $\text{nova\_solucao} \neq \text{NIL}$  then
7     if  $\text{melhor\_solucao} = \text{NIL} \vee \text{custo}(\text{melhor\_solucao}) >$ 
         $\text{custo}(\text{nova\_solucao})$  then
9        $\text{melhor\_solucao} \leftarrow \text{nova\_solucao};$ 
10       $\text{soma\_melhor\_solucao} \leftarrow \text{custo}(\text{nova\_solucao})$ 
11    else
12      if  $\text{custo}(\text{melhor\_solucao}) = \text{custo}(\text{nova\_solucao}) \wedge$ 
           $\text{soma}(\text{nova\_solucao}) < \text{soma\_melhor\_solucao}$  then
13         $\text{melhor\_solucao} \leftarrow \text{nova\_solucao};$ 
14         $\text{soma\_melhor\_solucao} \leftarrow \text{custo}(\text{nova\_solucao})$ 
15    end
16  end
17 end

```

O Algoritmo 1 parte de um vértice inicial para cada subconjunto em S e realiza um Busca em Largura (BFS) como busca no espaço de soluções. É verificado se a solução obtida por cada BFS tem menor custo (menos subconjuntos / voluntários) e como critério de desempate se possui a menor soma dos índices dos voluntários obtidos na solução.

1.2. Programação dinâmica (PD)

A base do algoritmo de programação dinâmica foi a do força-bruta. Identificou-se primeiramente a existência de subproblemas. Supondo a solução ótima $OPT(U)$ para um universo U de focos seja a sequência de voluntários $S^* = \{S_1, S_2, \dots, S_k\}$. Para um dado momento i do algoritmo ótimo, há um universo restante a ser coberto U^i que deve ser coberto pelos conjunto $S^i = \{S_i, S_{i+1}, \dots, S_k\}$, de maneira que o problema original pode ser resolvido de forma recursiva tal que $S^* = \{S_1, S_2, \dots, S_{i-1}\} \cup S^i$.

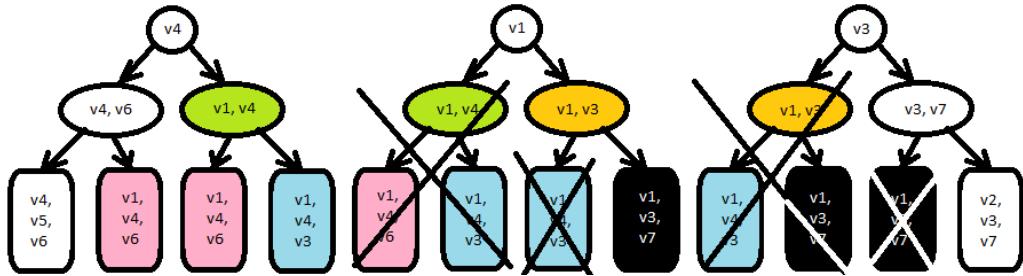


Figura 3. Nodos repetidos (coloridos)

Para utilização de técnicas de programação dinâmica é necessário também que haja sobreposição das soluções dos subproblemas. Neste caso, percebe-se que, no algoritmo de força-bruta há uma clara repetição de estados no conjunto de busca solução quando a BFS é chamada para cada vértice (Figura 3).

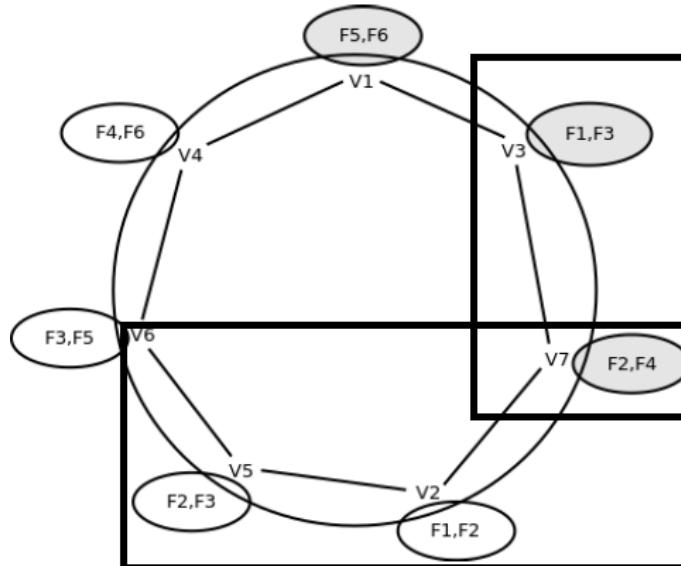


Figura 4. Exemplo de dois estados com a mesma cobertura resultante $\{F1, F2, F3, F4\}$

Outra computação redundante vislumbrada seriam os conjuntos cobertos resultantes de cada nodo, já que em um dado nível de profundidade da busca um determinado conjunto de elementos (exemplo Figura 4). Porém, as adjacências destes estados são diferentes, o que torna a utilização de uma estratégia de programação dinâmica não trivial.

Assim sendo, foi adotada uma estratégia de memorização de estados em que, quando um sucessor de um estado irá ser explorado, verifica-se se o mesmo já se encontra em uma lista de estados explorados. Se não estiver, este estado entra para a lista e é então verificado na BFS.

Algoritmo 2: Algoritmo ZikaZeroAnelDual programação dinâmica

Result: $\text{melhor_solucao} \vee \text{NIL}$

```

1  $\text{melhor\_solucao} \leftarrow \text{NIL};$ 
2  $\text{soma\_melhor\_solucao} \leftarrow 0;$ 
3  $\text{explorados} \leftarrow \text{ListaVazia}();$ 
4 foreach  $\text{subconjunto} \in S$  do
5    $\text{novo\_vertice} \leftarrow \text{subconjunto};$ 
6    $\text{nova\_solucao} \leftarrow \text{dyn\_BFS}(\text{novo\_vertice}, \text{explorados});$ 
7   if  $\text{nova\_solucao} \neq \text{NIL}$  then
8     if  $\text{melhor\_solucao} = \text{NIL} \vee \text{custo}(\text{melhor\_solucao}) > \text{custo}(\text{nova\_solucao})$  then
9        $\text{melhor\_solucao} \leftarrow \text{nova\_solucao};$ 
10       $\text{soma\_melhor\_solucao} \leftarrow \text{custo}(\text{nova\_solucao})$ 
11    else
12      if  $\text{custo}(\text{melhor\_solucao}) = \text{custo}(\text{nova\_solucao}) \wedge \text{soma}(\text{nova\_solucao}) < \text{soma\_melhor\_solucao}$  then
13         $\text{melhor\_solucao} \leftarrow \text{nova\_solucao};$ 
14         $\text{soma\_melhor\_solucao} \leftarrow \text{custo}(\text{nova\_solucao})$ 
15      end
16    end
17  end
18 end

```

O Algoritmo 2 de programação dinâmica proposto começa com uma lista vazia de estados explorados e passa como parâmetro para a BFS executada em cada vértice; cada novo estado encontrado por qualquer uma das BFS é inserido na lista como sendo a coleção de subconjuntos explorados até o momento. Caso ele já esteja na lista, não há necessidade de redundância em sua computação. Cabe dizer que a dyn_BFS utilizada é apenas uma sobrecarga do método BFS para lidar com o parâmetro adicional *explorados*.

1.3. Algoritmos gulosos

Um algoritmo gúloso é aquele que sempre escolhe o que parece melhor no momento, em outras palavras, é feita uma escolha ótima local e espera-se que ela seja a escolha ótima global para o problema. Entretanto, caso a solução retornada não seja ótima (como foi o caso) chamamos de heurística gúloso ou heurística aproximativa (caso possua uma razão de aproximação conhecida).

Algoritmo 3: Algoritmo guloso genérico

```
1 candidados  $\leftarrow$  {conj. de todos os elementos};  
2 solucao  $\leftarrow$  NIL;  
3 while candidados  $\neq \emptyset$  do  
4   c  $\leftarrow$  escolhe(estado);  
5   if viavel(solucao + c) then  
6     solucao  $\leftarrow$  solucao + c;  
7     candidados.remove(c)  
8   end  
9 end  
10 return solucao
```

Em todos os casos, primeiro deve-se mostrar que existe uma estratégia de escolha local gulosa que ofereça uma solução viável para o problema. De forma objetiva, deve existir uma função **escolhe**(*{candidados}*) de forma que o algoritmo possa ser escrito sob a forma do Algoritmo 3.

A estratégia de escolha gulosa utilizada é bastante simples. Verificam-se, como candidatos, os sucessores do estado atual (que serão exatamente dois, ou caso tenham sido explorados $n - 1$ subconjuntos do anel, será apenas o último subconjunto). Escolhe-se o candidato que, ao ser incorporado, mais aumenta a cobertura atual de elementos. Em caso de empate, é escolhido o subconjunto com o menor índice. O estado inicial é sempre o primeiro subconjunto, pois ele, pela definição do problema, maximiza os focos iniciais (já que cobre dois focos) e possui o menor índice.

Embora o método de escolha seja de fácil implementação, verificou-se experimentalmente que esta escolha não é ótima. Apesar de produzirem soluções corretas, nem sempre serão as de menor número de subconjuntos possível.

2. Análise de complexidade (Exercício 2)

Para fins de análise utilizaremos a notação *O-grande* para avaliar de pior caso dos algoritmos propostos.

2.1. Força-bruta (FB)

Sendo coerente com os conjuntos já utilizados n será o número de subconjuntos / voluntários $\in S$ e $m = |V|$ o número de laços de amizade entre os voluntários.

Complexidade de tempo: são realizadas n vezes a BFS que por sua vez tem complexidade $O(n + m)$. Desta forma, a complexidade do algoritmo será $O(n^2 + nm)$.

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], onde b é o fator de ramificação (ou *branching factor*) do grafo.

Neste caso, temos que o valor de $b = 2$ bem definido, já que este é o grau de adjacência de um dado estado. Desta forma, a complexidade final será $2 * O(n^2 + nm) = O(n^2 + nm)$.

2.2. Programação dinâmica (PD)

O algoritmo de PD é o mesmo do força-bruta com memorização de estados entre as n BFSs.

Complexidade de tempo: apesar de, em uma análise amortizada o caso médio tender a reduzir a complexidade, no pior caso sempre um estado novo será descoberto (que na prática não acontece), mantendo a complexidade de $O(n^2 + nm)$.

Complexidade de espaço: a estratégia de memorização utilizada impõe um limite superior aos estados memorizados de n vezes o espaço de uma BFS. Todavia, este também é o mesmo custo de memorizar os custos de n BFS como ocorre no força-bruta, portanto a complexidade de espaço é $O(n^2 + nm)$.

2.3. Algoritmos guloso

O algoritmo guloso não necessita de busca em largura (BFS) e segue o padrão descrito no Algoritmo 3.

Complexidade de tempo: função *escolhe()* que compara apenas 2 candidatos, sendo $\tilde{O}(1)$, esta função é chamada n vezes para a construção incremental da solução. A complexidade resultante é $O(n)$.

Complexidade de espaço: é necessária uma lista dos relacionamentos para indexar os sucessores que serão avaliados a cada iteração da *escolhe()*, desta forma a complexidade resultante será $O(m)$.

3. Avaliação experimental

3.1. Entradas utilizadas e execução de testes (Exercício 4)

Foram realizados testes na máquina do DCC Araguaia para dois conjuntos de entradas:

1. as disponibilizadas pelo monitor (ZikaZeroAnelDual.zip) para verificar a corretude das respostas.
2. entradas geradas por um *script* desenvolvido para testar o pior caso dos algoritmos.

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

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

Esse script sempre gera $n + 1$ focos, para n voluntários, sendo que são necessários todos os voluntários para cobrir todos os focos.

Arquivo	N	Saída esperada FB e		Saída GL	
		PD	esperada	FB	PD
in	7	1 3 7	2 4 5 6 7	1 3 7	1 3 7
in2	9	1 6 8 9	1 2 6 7 8 9	1 6 7 8 9	1 6 7 8 9
in3	9	1 6 8 9	1 2 6 8 9	1 6 8 9	1 6 8 9
in4	9	1 6 8 9	1 6 8 9	1 6 8 9	1 4 6 7 9
in5	17	12 13 14 15 16	1 2 3 4 5 6	12 13 14 15 16	1 2 3 4 5 16 17

Figura 5. Saidas para o primeiro grupo de entradas

Os resultados dos três algoritmos se foram semelhantes aos esperados pelas entradas fornecidas pelo monitor (Figura 5), com exceção da entrada "in2", porém acredita-se ser um erro da saída esperada 1 6 8 9, já que esta não é conectada como prevê o problema.

No caso do algoritmo guloso, na entrada "in2" o guloso implementado encontrou uma resposta melhor (1 6 7 8 9) do que a esperada (1 2 6 7 8 9). Já na entrada "in4" o algoritmo proposto teve uma resposta (1 4 6 7 9) pior que a esperada (1 6 8 9). Este fato é atribuído ao uso de diferentes estratégias para a função *escolhe()*.

3.2. Comparação entre resultado teórico esperado e custo real (Exercício 5)

Arquivo N	Comp.	Guloso (GL)		Programacao Dinamica (PD)		Forca-Bruta	
		Tempo exec (s)	Expansões	Complexidade	Tempo exec (s)	Expansões	Tempo exec (s)
v10	10	0,000	17		0,003	170	0,014
v30	30	0,002	57	$O(n^2 + nm)$	0,053	1710	0,640
v50	50	0,005	97		0,228	4850	5,267
v70	70	0,012	137		0,678	9590	22,170

Figura 6. Saídas para o primeiro grupo de entradas

Nas entradas do script (Figura 6), foram executados três vezes cada algoritmo para verificar os tempos de execução (em segundos) para entradas de tamanho $n = \{10, 30, 50, 70\}$. Também foram computadas as expansões de nós nos grafos durante as buscas por solução e colocadas as complexidades esperadas.

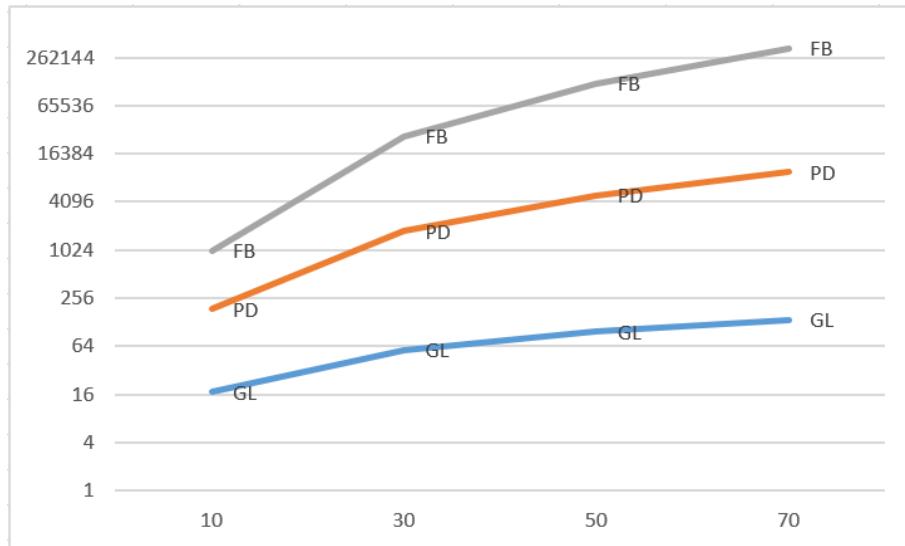


Figura 7. #nós expandidos pelos algoritmos em escala logarítmica (base 2)

Nota-se a dominação assintótica dos algoritmos, em uma escala logarítmica, dos nós expandidos (Figura 7). Como esperado, o algoritmo guloso (GL) expandiu menos nós que o de programação dinâmica (PD), que teve menos nós do que o força-bruta (FB). Outro fato esperado é o de que todas as curvas em escala logarítmica são decrescentes, o que demonstra um comportamento polinomial de todos eles.

Referências

- Russell, S. and Norvig, P. (2009). *Artificial intelligence: a modern approach*. Prentice Hall, 3rd edition.
- Shuai, T.-P. and Hu, X.-D. (2006). Connected set cover problem and its applications. In *Algorithmic Aspects in Information and Management*, pages 243–254. Springer.
- Vazirani, V. V. (2001). *Approximation algorithms*. Springer Science & Business Media.



Trabalho Prático 02- Paradigmas

1. Algoritmos Propostos

Para esse problema temos:

- Um grafo em anel 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.

Considere:

- $n = |V|$;
- $m = |A|$;
- $r = |F|$;

E numere os vértices no grafo em anel sequencialmente, de forma horária ou anti-horária:

- $w_0 = v_i \in V: i \in [0, n) - w_0$ é algum vértice em V ;
- $w_1 = v_j \in V: \exists(w_0, v_j) \in A - w_1$ é um dos dois vértices em V conectados a w_0 ;
- $w_i = v_k \in V: \exists(w_{i-1}, v_k) \in A, w_i \neq w_{i-2}$ – Os outros vértices serão ordenados seguindo o sentido definido por (w_0, w_1) .

Busca por Força-Bruta

Na abordagem por força bruta, testa-se todas as opções de conjuntos conexos possíveis no grafo $G(V, A)$, enquanto se verifica se eles são válidos (incluem todos os pontos de foco).

ForcaBrutaZikaAnel (graph)

```
1. smallest_set_size = n + 1
2. For i = 0 : n-1:
3.   For j = i : i + n - 1:
4.     If ChecarSeTodosFocos(graph, i, j):
5.       If j - i + 1 < smallest_set_size:
6.         smallest_set_size = j - i + 1
7.         smallest_set = (i, j)
8.       Break;
9. Return smallest_set
```

ChecarSeTodosFocos (graph, initial, final)

```
1. focuses = {}
2. For i = initial:final:
3.   current_node = graph.node(i)
```

4. Union(focuses, GetFocuses(corrente_node))
5. If focuses.size == r:
6. Return True
7. Return False

Programação Dinâmica

Na abordagem de programação dinâmica, pode-se criar um novo grafo $G'(V', A')$ que tem $|V'| = 2 \cdot |V| - 1$, tal que ele é um grafo linear com vértices:

$$V' = \{w_1, w_2, \dots, w_n, w_1, w_2, \dots, w_{n-1}\}$$

E conectado nessa ordem. Note que todo subgrafo conectado de $G(V, A)$ está contido em $G'(V', A')$. Acha-se, então, a melhor solução incluindo apenas os primeiros i vértices. Essa solução vai ser, ou a melhor solução completa que inclua o vértice i , ou a melhor solução para $i - 1$:

$$OPT(i) = \min \left(\begin{array}{l} OPT(i - 1) \\ focos_terminando_em(i).tamanho \end{array} \right)$$

Utilizando o conjunto de focos obtido para $i - 1$ ($focos_terminando_em(i - 1)$), pode-se calcular facilmente $focos_terminando_em(i)$.

ProgramacaoDinamicaZikaAnel (graph)

1. opt = vetor int[2n]
2. opt[0] = +inf
3. opt_set = vetor pair[2n]
4. opt_set[0] = NULL
5. focuses = struct{focuses}
- 6.
7. For i = 1 : 2n - 1:
8. cost_with_i = inf
9. focuses = ObterFocos(focuses, i)
10. if focuses.num(i) == r:
11. cost_with_i = focuses_set.size
- 12.
13. opt[i] = min(opt[i-1], cost_with_i)
14. If cost_with_i < opt[i-1]:
15. opt_set[i] = (focuses.start_node, focuses.end_node)
16. Else
17. opt_set[i] = opt_set[i-1]
- 18.
19. Return opt_set[2n-1]

ObterFocos(focuses_set, i)

1. node = graph.node(i)
- 2.
3. focuses_set.end_node++
4. For focus = GetFocuses(node):
5. If focuses_set[focus] == 0:

```

6.     focuses_set.num++
7.     focuses_set[focus] += 1
8.
9. // While all focuses in initial node are redundant, remove this node
10. While True:
11.     node = graph.node(focuses_set.start_node)
12.     For focus = GetFocuses(node):
13.         If focuses_set[focus] < 2:
14.             Break;
15.         For focus = GetFocuses(node):
16.             focuses_set[focus]--
17.             focuses_set.start_node++
18.
19. focuses_set.size = focuses_set.end_node - focuses_set.start_node + 1

```

Algoritmo Guloso

Para o algoritmo gulosso, se começa com um conjunto de apenas um nó 0 e com acesso a seus focos de mosquito ($set(i,j) = (0,0)$). A cada iteração do algoritmo uma escolha gulosso é realizada. Nela, se decide entre pegar o próximo nó, ($j = j + 1$), ou se reduzir o conjunto de nós, avançando o começo desse conjunto ($i = i + 1$).

$$set(i,j) = \begin{cases} set(i,j+1) & \text{se } |focos(i,j)| < r \\ set(i+1,j) & \text{se } |focos(i,j)| = r \end{cases}$$

Se repete esse processo até dar a volta no grafo, sempre salvando a melhor resposta completa (em que $|focos(i,j)| = r$) até o momento.

GulosoZikaAnel (graph)

```

1. focuses = vector zeros[r]
2. num_focuses = 0
3. smallest_set_size = n
4. j = i = 0
5. initial_node = graph.node(0)
6. AdicionarFocos (focuses, num_focuses, initial_node):
7.
8. While i < n:
9.     If num_focuses < r:
10.        j++
11.        final_node = graph.node(j % n)
12.        AdicionarFocos (focuses, num_focuses, final_node)
13.    Else:
14.        If j - i < smallest_set_size:
15.            smallest_set_size = j - i
16.            smallest_set = (i, j % n)
17.            If smallest_set_size == 0:
18.                Break;
19.
20.        initial_node = graph.node(i % n)
21.        RemoverFocos (focuses, num_focuses, initial_node)

```

- 22.
23. i++
24. Return smallest_set

AdicionarFocos (focuses, num_focuses, node)

1. For focus = GetFocuses(node):
2. If focuses[focus] == 0:
3. num_focuses++
4. focuses[focus] += 1

RemoverFocos (focuses, num_focuses, node)

1. For focus = GetFocuses(node):
2. focuses[focus] -= 1
3. If focuses[focus] == 0:
4. num_focuses--

2. Análise de Complexidade

Busca por Força-Bruta

O algoritmo de força-bruta tem ordem de complexidade temporal $O(n^3)$, o que pode ser visto pois a função ChecarSeTodosFocos tem complexidade $O(n)$ e ela é executada $O(n^2)$ vezes.

Esse algoritmo, no entanto, para um r fixo e um $n \gg r$, tem caso médio $O(n \cdot E(x)^2)$, sendo x o número mínimo de elementos num conjunto completo. Isso ocorre pois o algoritmo para, em cada loop, após achar um conjunto completo. O valor esperado de x será:

$$E(x) = \sum_{k=1}^r \frac{r}{k} = r \cdot \sum_{k=1}^r \frac{1}{k} = r \cdot H(r)$$

Sendo $H(r)$ o r -ésimo número harmônico. O caso médio será, então, $O(n \cdot E(x)^2) = O(n \cdot r^2 \cdot H(r)^2)$. Já para $r \approx n$, ou $r \geq n$, $E(x) \approx n$. Isso implica em que $O(n \cdot E(x)^2) = O(n^3)$.

Esse algoritmo tem complexidade espacial $O(r)$, sendo a única estrutura de tamanho não constante salva por ele o set contendo os focos de mosquito na função ChecarSeTodosFocos.

Programação Dinâmica

Esse algoritmo tem ordem de complexidade temporal $O(n)$, mesmo se tendo um loop na função ProgramacaoDinamicaZikaAnel que executa a função ObterFocos, que tem complexidade temporal $O(n)$, $O(n)$ vezes. Isso pode ser verificado fazendo uma análise amortecida dessas funções. Note que $end_node \geq start_node$ e que end_node aumenta em apenas uma unidade por execução da função. Então, embora uma única execução de ObterFocos possa ser $O(n)$, $2 \cdot n$ execuções seguidas também estão limitadas a essa mesma complexidade, desde que end_node e $start_node$ não sejam alterados fora dela.

A complexidade espacial desse algoritmo é $O(n + r)$, dados pelas estruturas opt e opt_set , que são $O(n)$, e por $focuses_set$, que é $O(r)$.

Algoritmo Gúloso

O algoritmo guloso também possui complexidade temporal $O(n)$. Note que as funções AdicionarFocos e RemoverFocos possuem complexidade temporal $O(1)$, pois cada vértice tem acesso a apenas dois focos diferentes. Note também que o valor de j está limitado por i ($j < i + n$), pois um conjunto com n vértices sempre vai ter acesso a todos os focos, e o valor de j só cresce se não são todos os focos que já tem acesso. Então, como $i < n$ e ou i ou j necessariamente aumentam em uma unidade por loop, esse loop será executado $O(n)$ vezes.

A complexidade média desse algoritmo tem complexidade temporal $O(n + E(x))$, pois $j \leq i + x < n + x$. Esse valor, assim como para o caso de Força-Bruta, para $r \ll n$, $O(n + E(x)) = O(n + r \cdot H(r))$. Para $r \approx n$: $O(n + E(x)) = O(n + n) = O(n)$.

O algoritmo guloso possui complexidade espacial $O(r)$, dada pelo vetor *focuses*, que tem tamanho r e é a única estrutura de tamanho não constante usada no algoritmo.

3. Implementação

Algoritmos implementados em *Python*.

4. Execução de Testes

Testes foram executados para variações dos parâmetros:

- $n = |V|$, número de voluntários;
- $r = |F|$, número de focos de reprodução do mosquito.

Para r variável, os testes foram realizados mantendo-se n constante. Os testes para variações de n foram realizados para dois casos distintos: matendo-se r constante; utilizando um valor de r aleatório. Para todos os testes, para cada valor da variável analisada (n ou r), foram realizadas várias execuções e o valor médio delas foi usado.

Os testes são apresentados na próxima sessão, juntamente com sua análise. Não foram realizados testes específicos para a variação de m pois, em um grafo anel, $m = n$.

5. Comparação da Análise de Complexidade com os Testes

Teste para variação de r

1. Busca por Força-Bruta

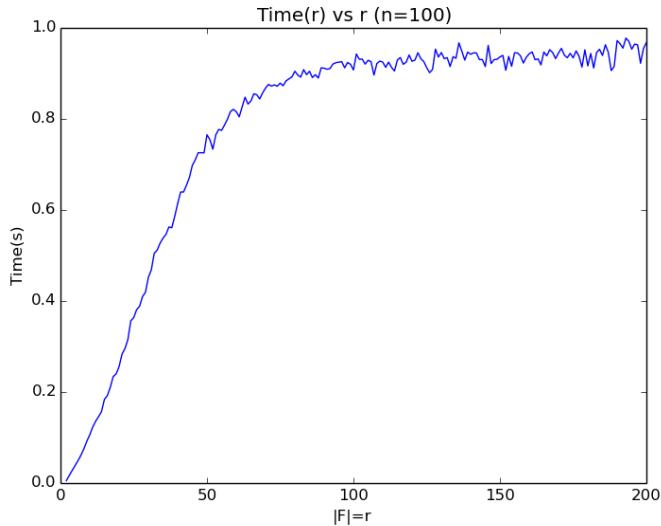


Figura 1 – Busca por Força-Bruta - Tempo x r

Esse gráfico demonstra como, para valores pequenos de r ($r \ll n$), a complexidade média do algoritmo é dependente de r , com complexidade $O(n \cdot r^2 \cdot H(r)^2)$. No entanto, para valores grandes, a complexidade do algoritmo é constante em r , seguindo $O(n^3)$.

2. Programação Dinâmica

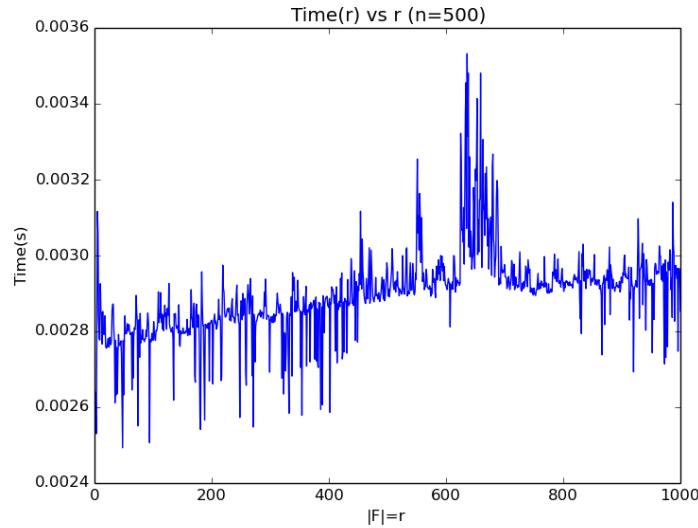


Figura 2 - Programação Dinâmica - Tempo x r

Esse gráfico mostra como o tempo de execução do algoritmo de programação dinâmica é constante em função de r , assim como analisado.

3. Algoritmo Gúloso

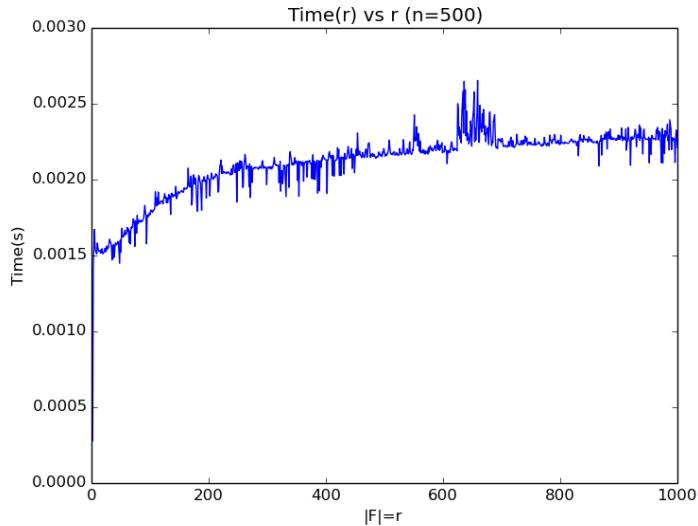


Figura 3 - Algoritmo Guloso - Tempo x r

Nesse gráfico, pode-se ver um aumento inicial no tempo de execução do algoritmo e, após isso, ele vira constante em função de r . Isso se dá pois, assim como já explicado, para $r \ll n$, o tempo médio do algoritmo é dependente de r : $O(n + E(x)) = O(n + r \cdot H(r))$. Mas para um valor de r não muito baixo, $O(n + E(x)) \approx O(n)$.

Testes para variação de n

1. Busca por Força-Bruta

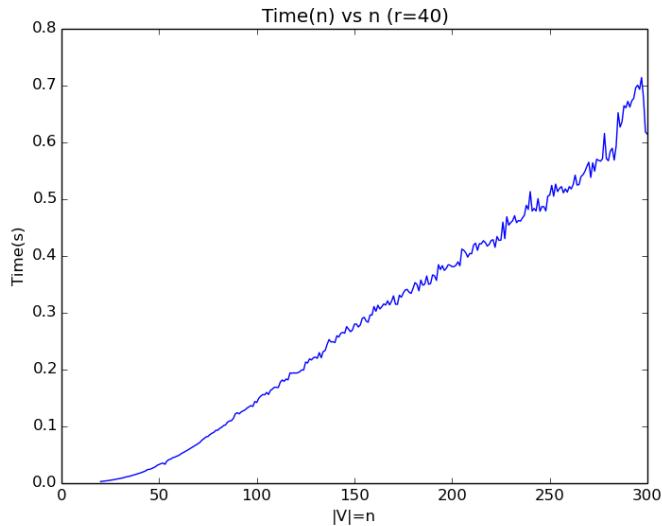


Figura 4 - Busca por Força-Bruta - Tempo vs n (r fixo)

Assim como analisado, para um r fixo, o tempo necessário para executar esse algoritmo tem, inicialmente, complexidade $O(n^3)$. Conforme n aumenta, r se torna muito menor que n ($r \ll n$), então $O(n \cdot E(x)) = O(n)$, para esse valor de r constante. Isso pode ser claramente visto no gráfico: para valores de n altos, o tempo cresce linearmente com n .

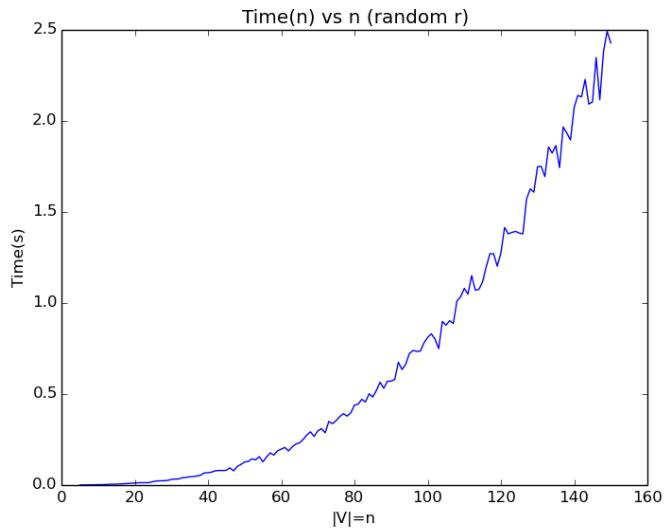


Figura 5 - Busca por Força-Bruta - Tempo vs n (r randômico)

Para um valor de r randômico, distribuído uniformemente entre $[1; 2 \cdot n]$, não vale a afirmação $r \ll n$, então a função tem sempre tempo médio $O(n^3)$.

2. Programação Dinâmica

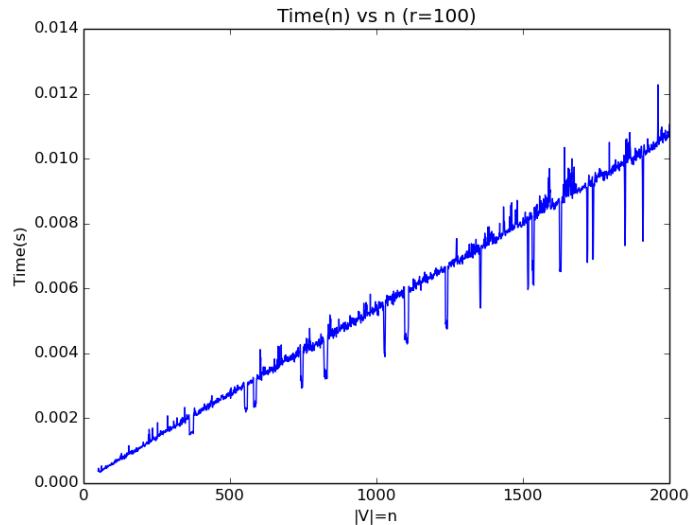


Figura 6 - Programação Dinâmica - Tempo vs n (r fixo)

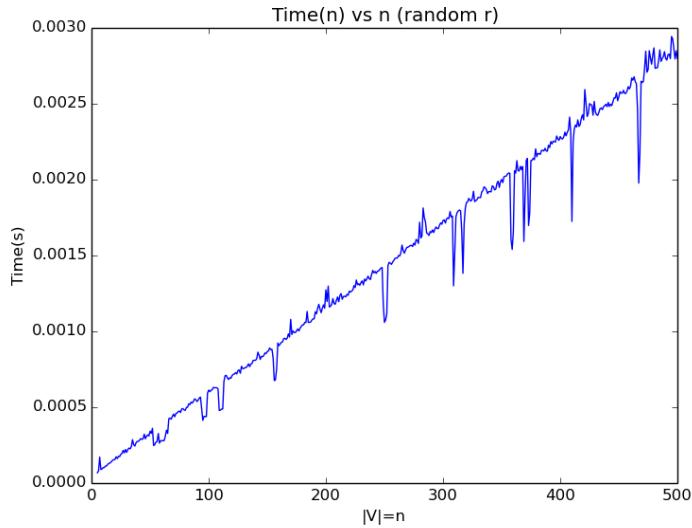


Figura 7 - Programação Dinâmica - Tempo vs n (r randômico)

Como pode ser observado nos gráficos e como foi analisado, para o algoritmo de programação dinâmica, tanto para um r fixo ou randômico, a complexidade temporal depende apenas de n , sendo linear nesse valor ($O(n)$).

3. Algoritmo Guloso

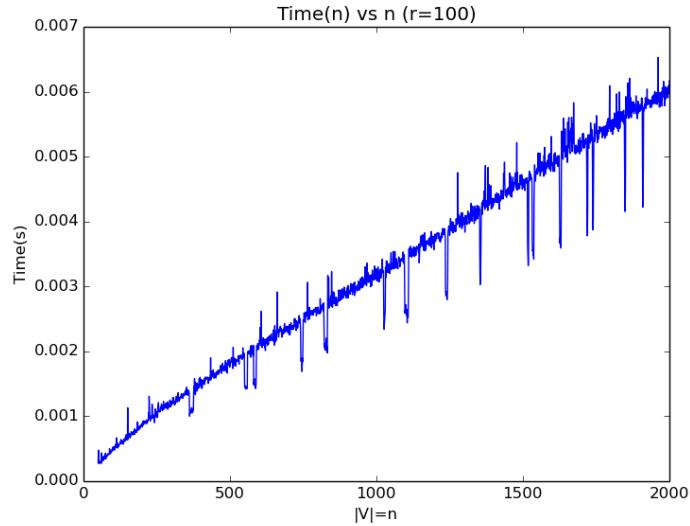


Figura 8 - Algoritmo Guloso - Tempo vs n (r fixo)

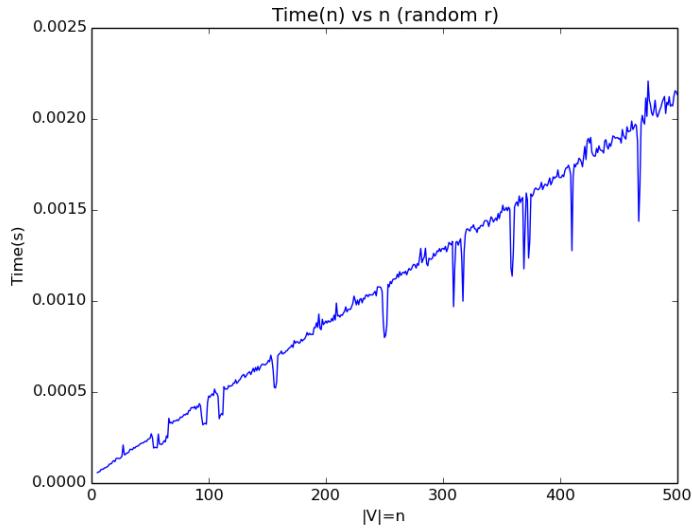


Figura 9 - Algoritmo Guloso - Tempo vs n (r randômico)

No algoritmo guloso, assim como no de programação dinâmica, o tempo necessário cresce linearmente com n , seja para um r fixo ou randômico. Note, no entanto, que para um r fixo:

- Para valores baixos de n baixos: enquanto $r \approx n$, $\text{tempo} \propto a \cdot n + b \cdot E(x) \approx (a + b) \cdot n$
- Para $r \ll n$: $\text{tempo} \propto a \cdot n + b \cdot E(x) \approx a \cdot n$,

Para $r \ll n$, então, tem-se uma constante multiplicando n menor, o que pode ser visto no gráfico com r fixo acima.

Projeto e análise de algoritmos - Trabalho Prático 2

Evandro Caldeira¹

¹UFMG

evandro@dcc.ufmg.br

1. Exercício 1

O algoritmo de força bruta pode ser visto na listagem 1. A função *subset* gera todas as combinações possíveis de subconjuntos da lista passada por parâmetro. Partindo dessa lista as soluções são avaliadas e analisadas para se encontrar a menor solução.

Algorithm 1: Algoritmo de força bruta

```
Function FocaBruta(nodes)
    // variaveis de entrada voluntarios = read_file()
    focus = read_file()
    //
    // variaveis de controle
    melhor_valor = ∞
    melhor_solucao = ∅
    for solucao In subset(voluntarios) do
        if is_valid(solucao, focus) then
            valor_atual = evaluate(solucao)
            if valor_atual < melhor_valor then
                melhor_valor = valor_atual
                melhor_solucao = solucao
    return melhor_solucao
```

O algoritmo 2 para solução gulosa escolhe primeiro os conjuntos que cobrem mais focos do Zika. Para fazer isso o primeiro laço percorre todos os focos para saber quais são acessados por mais voluntários. O segundo laço utiliza essa lista para escolher primeiro os focos de acessos de mais voluntários. Assim que todos os focos estiverem cobertos e houver uma rede de voluntários a função retorna a solução.

Algorithm 2: Algoritmo guloso

```
Function Guloso(grafo)
    voluntarios = grafo.voluntarios
    focus = grafo.focus
    solucao = ∅
    for voluntario In voluntarios do
        for foco In focus[voluntario] do
            total_focus[foco] += 1
    for foco In total_focus ordenado decrescente do
        solucao += foco
        if valida(solucao) then
            return solucao
    return ∅
```

O algoritmo de programação dinâmica 3 realiza a busca em etapas divididas pelo tamanho do subconjunto de pesquisa. Na iteração 1 são explorados os conjuntos de tamanho 1, na iteração 2 de tamanho 2 e na n de tamanho n . Como são pesquisados conjuntos temos que $1, 2, 3 = 3, 2, 1$. Essa estratégia é importante para reduzir o espaço de busca. Outro fator importante é a reutilização de soluções. Por exemplo tendo encontrado $A = 2, 3$ na iteração 2, na iteração 3 esse resultado será reutilizado.

Algorithm 3: Algoritmo dinâmico

```
Function dinamico(grafo)
    focus = grafo.focus
    voluntarios = grafo.voluntarios
    solucao = ∅
    for tamanho_conjunto=1 To length(voluntarios) do
        if s[1:] In cache then
            solucao_parcial = cache[s[1:]]
            solucao = s[0] + solucao_parcial
        else if s[0: n-1] In cache then
            solucao_parcial = cache[s[0: n-1]]
            solucao = s[n] + solucao_parcial
            cache[s] = solucao
        if valida(solucao) then
            return solucao
    return ∅
```

2. Exercício 2

Para complexidade temos n como o número de voluntários e m o número de focus de infestação. Para as listas de adjacências tomemos como exemplo o par de vértices (v_i, v_j) ligados entre si. Em nossa implementação teremos uma lista v_i com uma entrada para v_j e uma lista v_j com um entrada para v_i . Essa escolha foi feita para facilitar a implementação e a busca dos vértices adjacentes sem a necessidade de pesquisar várias listas. Os valores de complexidade estão listados na Tabela 1.

	Força Bruta	Guloso	Programação Dinâmica
Temporal	$O(V^2 * F)$	$O(V * F)$	$O(V^3)$
Espacial	$O(V^2)$	$O(V)$	$O(V * F)$

Table 1. Análise de complexidade

3. Exercício 3

Implementação

4. Exercício 4

Na Figura 1 temos os tempos de execução para entradas até o tamanho 100. Na tabela 2 temos todos os tempos de execução.

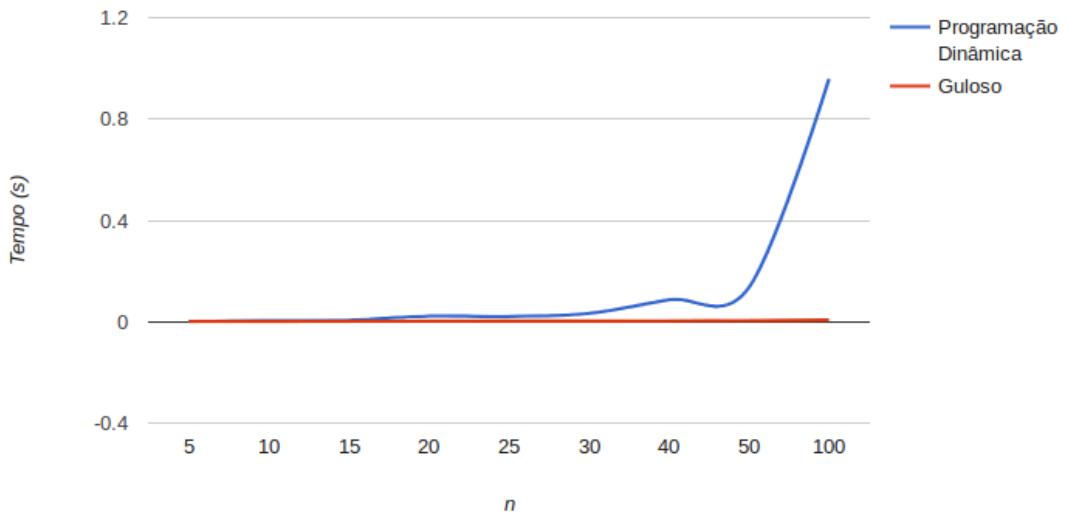


Figure 1. Tempos de execução

n	Programação Dinâmica	Guloso	Força Bruta
5	0.001160621643	0.001176357269	0.001223564148
10	0.004446268082	0.0009996891022	0.007307767868
15	0.006294727325	0.001460075378	0.2612810135
20	0.02223062515	0.002879619598	10.68785667
25	0.0205950737	0.002645492554	
30	0.03279733658	0.00226855278	-
40	0.08680510521	0.002857923508	-
50	0.1354784966	0.00403213501	-
100	0.9586074352	0.007151842117	-

Table 2. Tempos de execução

5. Exercício 5

Comparando os resultados vistos na figura 1 com as análises de complexidade da tabela 1 podemos ver empiricamente que o algoritmo guloso é mais rápido do que os demais.

Apesar de ser mais rápido o guloso nem sempre entrega a solução ótima enquanto que a implementação de força bruta e programação dinâmica sim. A implementação de força bruta se mostrou a mais ineficiente pois sempre são investigados todos os estados possíveis. O guloso é o mais rápido dos três e quando não entrega a solução ótima chega próximo a ela com uma razão de aproximação $H(x) = 2$.

Projeto e Análise de Algoritmos

Trabalho Prático - Módulo 3

ZicaZeroAnelDual

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

Considerando um grafo em formato de anel $G(V, A)$ não-direcional e sem pesos nas arestas. Onde cada vértice possui uma função de mapeamento $R : v \rightarrow [f_1..f_n]$ e $|R(v)| = 2$. Devemos encontrar um subgrafo conexo G' que cubra todos os focos do conjunto F .

1.1 Força Bruta

Para esta abordagem procuramos gerar todas as combinações de vértices que cobrem os focos desejados, seguindo o mesmo sentido de locomoção no gráfico. Abordagem que só é possível pela topologia em anel do gráfico e suas restrições.

```
S ← {};
for v in V do
    p ← v;
    Xtemp ← {};
    Stemp ← {};
    while |xtemp| < |F| do
        Xtemp ← Xtemp + {R(p)};
        Stemp ← Stemp + {p};
        p ← p.adj;
        if p == v then
            | break;
        end
    end
    if |S| == 0 or |S| < |Stemp| then
        | S ← Stemp ;
    end
end
```

Algorithm 1: Força Bruta

1.2 Algoritmo Guloso

Nossa ideia de aproximação segue os padrões do algoritmo de força bruta e o algoritmo guloso para cobertura de conjuntos com algumas modificações. A primeira modificação é encontrar um subconjunto k de vértices iniciais que junto com seus adjacentes produzem uma melhor cobertura. Incluímos

também, outro passo gulosso onde a partir do momento em que conseguiu encontrar um caminho de tamanho V' , interrompendo a busca de todas soluções que extrapolam este tamanho.

```

 $V \leftarrow selectInitNode(G)[0 : k]$  ;
for  $v$  in  $V$  do
     $p \leftarrow v + vi$ ;
     $Xtemp \leftarrow \{\}$ ;
     $Stemp \leftarrow \{\}$ ;
    while  $|xtemp| < |F|$  do
         $Xtemp \leftarrow Xtemp + \{R(p)\}$ ;
         $Stemp \leftarrow Stemp + \{p\}$ ;
         $p \leftarrow p.adj$ ;
        if  $|S| > |Stemp|$  or  $p == v$  then
            | break;
        end
    end
    if  $|xtemp| == |F|$  and ( $|S| == 0$  or  $|S| < |Stemp|$ ) then
        |  $S \leftarrow Stemp$  ;
    end
end

```

Algorithm 2: Guloso

```

 $maxV \leftarrow 0$ ;
 $maxVCover \leftarrow \{\}$ ;
/* Seleciona dos vértices aquele que com seus vertices
adjacentes, da direita e esquerda, forme um
subconjunto com maior cobertura inicial. */ 
while  $v$  in  $V$  do
    /* inserção em ordem reversa por cobertura */
     $VCover \leftarrow \{v : R(v) + R(v.right)\}$  ;
end
return  $maxV$ ;

```

Algorithm 3: selectInitNodes

2 Questão 2

2.1 Complexidade Temporal

O algoritmo de Força Bruta sempre será $O(n^2)$ no pior ou melhor caso. O algoritmo guloso é $O(n^2)$ no pior caso e $O(n + c)$ no melhor caso com c equivalente ao tamanho do melhor caminho, isto quando o vértice inicial é equivalente ao nó inicial da solução ótima.

2.2 Complexidade Espacial

O algoritmo de Força Bruta será $O(n + m)$ (tamanho do grafo), pois somente a melhor solução é armazenada em memória. Para o algoritmo guloso o custo também é $O(n + m)$ seguindo as mesmas premissas do de força bruta.

3 Questão 3

Para este trabalho utilizei python 2.6 com o numpy. O Código é dividido em 3 arquivos principais:

- main.py - Interpretação dos arquivos de entrada e saída.
- ZicaZAnelDual.py - Com a classe que implementa os algoritmos para resolução do problema.
- graphutil.py - Arquivo com a classe responsável por representar o grafo e suas relações.

4 Questão 4

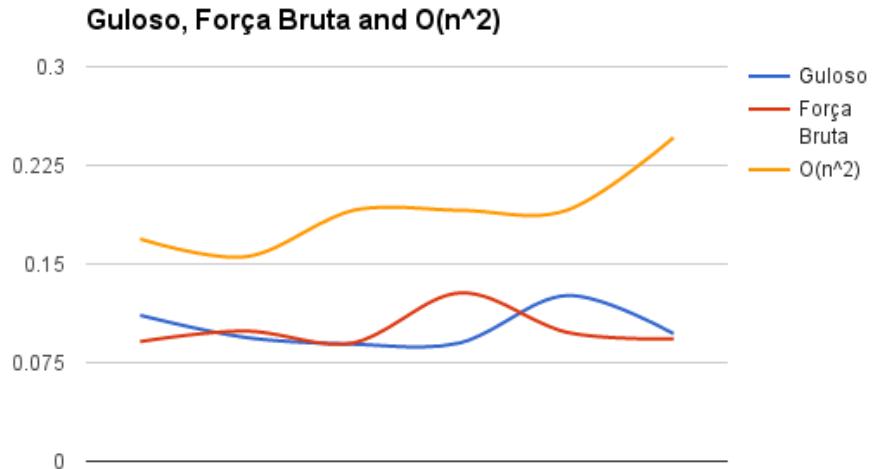


Figure 1: Comparativo de tempo.

O gráfico exposto na figura 1 é um comparativo de tempo entre os métodos implementados pelo tamanho da entrada. Como os métodos são todos com complexidade de tempo em função de n ou $|V|$, colocamos os seus tempos com uma função de tempo hipotética que segue o padrão $O(n^2)$.

5 Questão 5

O tempo esperado para execução do algoritmo de força bruta era de $O(n^2)$, porém, algumas otimizações propostas como evitar que o algoritmo continue depois que já fez um caminho que cobrisse todos os focos, tornou-o mais eficiente. No tempo geral de execução, o algoritmo guloso também foi melhor que o esperado pela análise assintótica e em comparação com o de força bruta. Em quase todos os cenários e obteve resultados ótimos, 3 dos 5 experimentos que rodamos. Porém, em uma das instâncias sua escolha gulosa o fez mais lento do que o algoritmo de força bruta otimizado.

Projeto e Análise de Algoritmos – Trabalho prático de paradigmas - 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 connected 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 of mosquito breeding spots and keep the resulting graph connected. Such solution is composed of three different algorithms along with their corresponding complexity function. Also several tests are run in order to analyse and compare their efficiency.

Resumo. Este documento apresenta a definição e resolução do problema connected 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. São propostos três algoritmos com diferentes paradigmas, juntamente com suas funções de complexidade e são realizados testes que analisam a eficiência de tais soluções.

1. Introdução

Este documento apresenta a resolução do trabalho prático de paradigmas proposto na disciplina de PAA, contendo a solução do problema ZikZeroZ implementada em C++ utilizando algoritmos em três paradigmas distintos: busca sistemática, ou força bruta (*brute-force*), algoritmo guloso e programação dinâmica. 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 se utiliza de um grafo representando uma rede de voluntários e laços de amizade, onde 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. Além disso, todas as entradas do problema correspondem a um grafo em anel, ou seja, um grafo G onde todo vértice tem grau 2.

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, e que $|R(v)| = 2 \forall v \in V$.

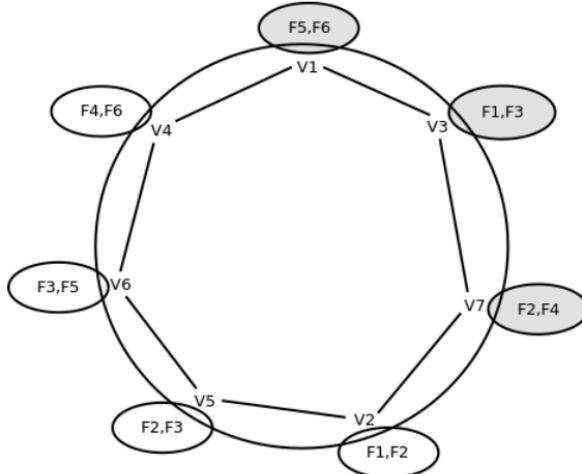


Figura 1. Instância de exemplo do problema ZikaZeroZ. Cada vértice corresponde a um voluntário e as arestas apresentam as relações entre os voluntários. Cada voluntário tem acesso a exatamente dois focos.

Neste cenário, o problema consiste em descobrir o menor número de voluntários que permitem acessar todos os focos do Zika Vírus. Além disso, os voluntários selecionados precisam formar um grafo conexo. 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 $R(v) \subseteq F$, com a propriedade de que $|R(v)| = 2 \forall v \in V$.

2.1. Redução para o problema do set-cover

De acordo com Cormen, Leiserson, Rivest, e 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.

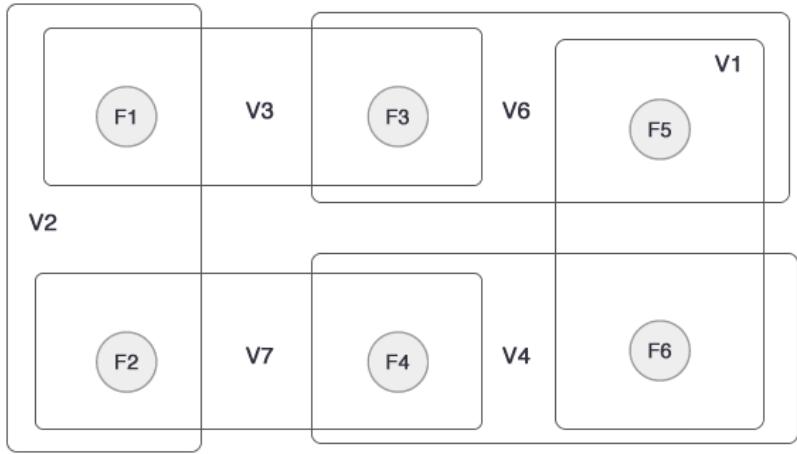


Figura 2. Representação do grafo da Figura 1 como uma instância do set-cover.
Cada vértice representa um conjunto e cada foco do Zika Vírus representa um elemento do universo. O vértice 5 foi omitido da imagem para facilitar a visualização.

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.

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.

$$X = \bigcup_{F \in C} F \quad \text{Eq. 1}$$

2.1.1 Casos particulares do set-cover

Como mostrado por Zhang, Gao, e Wu (2009), existem situações particulares onde o *set-cover* possui uma solução ótima que pode ser obtida em tempo polinomial. Existem ainda, casos onde é possível gerar soluções gulosas com razões de aproximação quase constantes, crescendo apenas com o logaritmo da entrada (Ren & Zhao, 2011). O trabalho apresentado neste relatório se baseia no fato de que o grafo que define o problema é um grafo em anel, o que reduz drasticamente a quantidade de possibilidades no espaço de soluções.

Embora não esteja no escopo deste trabalho analisar o motivo de tal redução, dada uma família de entradas específicas do problema, é fácil perceber que para um grafo em anel (onde o grau de cada vértice é exatamente 2), se um determinado vértice for escolhido como sendo parte da solução ótima do problema, existem apenas duas possibilidades de escolha para novos vértices, que corresponde a seguir o caminho do grafo no sentido horário ou anti-horário, dado que, ao entrar em um vértice por uma aresta a_1 , somente é possível sair do mesmo (sem repetir nenhum vértice já selecionado) pela aresta a_2 .

2.2. Modelagem do problema e definições iniciais

A modelagem do problema consiste em percorrer o espaço de soluções através de três paradigmas de programação diferentes: força bruta, programação gulosa e programação dinâmica. A solução obtida deverá ser ótima para os casos de força bruta e programação dinâmica, ao passo que o algoritmo guloso poderá apresentar resposta aproximada.

Dada a natureza da entrada do problema, a abordagem consistirá sempre em selecionar as soluções de forma que os vértices escolhidos formem um grafo conexo, eliminando a necessidade de se aplicar algoritmos de busca para garantir esta propriedade ao final do algoritmo.

Dados informados/conhecidos

- i) O grafo G é não direcionado e não possui pesos nas arestas.
- ii) G é um grafo em anel.
- iii) O conjunto de vértices V (voluntários).
- iv) O conjunto de arestas A (relações de amizade).
- v) O conjunto de focos F .
- vi) A relação entre os vértices e os focos, definida por R .
- vii) Cada voluntário acessa exatamente dois focos.

Suposições e/ou limitações

- i) O *set-cover* é um problema NP-Completo (Cormen, Leiserson, Rivest, & Stein, 2011) mas a família de entradas proposta permite que sejam obtidas soluções polinomiais.
- 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 as propriedades $v.l$ e $v.r$ que informam o vértice anterior e posterior (*left* e *right*), respectivamente.
- viii) A solução escrita na saída deve estar em ordem crescente de acordo com o índice do vértice.
- ix) Sempre que for citado, as operações de considerar o vértice seguinte e anterior correspondem a analisar o vértice adjacente no sentido horário e anti-horário, respectivamente.

2.3 Redução do espaço de buscas

Como já citado anteriormente, a família de entradas válidas para o problema gera uma redução drástica na quantidade de elementos no espaço de soluções e tirando proveito desta redução é possível construir algoritmos polinomiais.

A primeira redução drástica acontece na escolha dos vértices durante a busca por uma solução válida. Suponha que uma solução ótima seja o caminho $p: v_i \rightarrow v_j$, ou seja: o caminho que vai do vértice v_i até v_j e possui l vértices. Suponha ainda que os caminhos são apresentados de forma a serem percorridos no sentido horário. Durante a execução do algoritmo, seja um vértice selecionado v_k , se v_k faz parte de p então existem apenas as seguintes possibilidades:

- 1) $v_k = v_i$: o vértice selecionado é o primeiro do caminho p e neste caso a única possibilidade é considerar o vértice seguinte.
- 2) $v_k = v_j$: o vértice selecionado é o último do caminho p e neste caso a única possibilidade é considerar o vértice anterior.
- 3) $v_k | i < k < j$: o vértice selecionado está dentro do caminho. Neste caso há apenas duas possibilidades: (i) selecionar o próximo vértice; (ii) selecionar o vértice anterior.

Os passos mencionados acima são repetidos sistematicamente até que se tenha coberto todo o caminho p . Como Somente existe uma única possibilidade de escolha qualquer que seja o vértice selecionado, a árvore de busca não apresenta crescimento exponencial, reduzindo a complexidade do problema.

A segunda redução, que é vista mais especificamente como um tipo de poda, é baseada no fato de que cada voluntário tem acesso a exatamente dois focos. Neste caso existe um número mínimo de vértices que deverão estar contidos na solução ótima. Seja p_{min} o caminho mínimo que corresponde à solução ótima. Neste caso $|p_{min}| = \lceil r/2 \rceil$. Esta poda elimina a necessidade de se construir caminhos com menos vértices do que o mínimo requerido para a solução ótima uma vez que sabidamente tal combinação não levará a um resultado válido.

3. Resolução por força bruta

A resolução por força bruta é certamente a maneira mais direta de se atacar o problema. A estratégia consiste em percorrer sistematicamente todas as combinações possíveis de respostas e verificar a melhor delas. A seguir está apresentado um algoritmo geral que resolve o problema por força bruta.

```
1  FORCA-BRUTA-GERAL (G, F)
2      let best = G.V
3      for each combination V' of G.V that forms a connected path
4          if |V'| < |best|
5              best = V'
6
7      return best;
```

O algoritmo **FORCA-BRUTA-GERAL**(G, F) recebe como parâmetros um grafo G e o universo F a ser coberto e analisa sistematicamente todas as combinações válidas, procurando por aquela que possui menor cardinalidade. O algoritmo apresentado a seguir corresponde a uma especialização do algoritmo **FORCA-BRUTA-GERAL**(G, F) e resolve o problema ZikaZeroZ.

```

1  FORCA-BRUTA( $G, F$ )
2      let  $V = G.V$ 
3
4      for  $i = |F|/2$  to  $|V|$ 
5          for each  $v$  in  $V$ 
6              let  $S = \{v\}$ 
7              for  $j = 2$  to  $i$ 
8                   $S = S + \{\text{next vertex of } V\}$ 
9
10         if  $R(S) = F$            //fazemos (ab)uso da relação  $R$ 
11         return  $S$ 
```

O algoritmo mais específico **FORCA-BRUTA**(G, F) recebe como entrada um grafo de anel G e um universo F . O *loop* que começa na linha 4 informa o tamanho dos caminhos p que serão construídos em cada iteração. O *loop* da linha 5 passa por todos os vértices uma vez, escolhendo o vértice inicial do caminho sendo formado, e o *loop* da linha 7 adiciona os próximos elementos adjacentes ao caminho, até o limite máximo permitido para aquela iteração. Após formar um caminho, a linha 10 verifica se este cobre o conjunto de focos. Em caso positivo, este caminho é retornado e o algoritmo encerra sua execução. Neste caso, como começamos construindo os caminhos de forma incremental, a primeira solução conterá o menor caminho possível capaz de cobrir todos os focos.

De acordo com as premissas (iv) e (v), o algoritmo sempre irá apresentar uma solução para as entradas apresentadas. O algoritmo por força bruta proposto tem a característica de apresentar uma solução ótima. Para uma execução no pior caso, o *loop* da linha 4 será executado um total de $|V| - |F|/2$ vezes; Neste cenário, considere que $|F| = 2V$, que é o valor máximo de focos que é possível cobrir com $|V|$ voluntários onde cada voluntário tem acesso a exatamente dois focos. Dessa maneira, a linha 4 será executada uma única vez. Por outro lado, se $|F| = 3$, que é a quantidade mínima de focos que foge à solução trivial, a linha 4 será executada aproximadamente $|V|$ vezes.

Em contrapartida, a linha 5 sempre é executada $|V|$ vezes pois é preciso percorrer todos os vértices para formar os caminhos da solução; e a linha 7 é percorrida aproximadamente $|V|/2$ vezes. Sendo assim, escrevendo $|F|$ em função de V , a complexidade temporal do algoritmo é dada por:

$$f(n) = V \times V \times V/2 = O(V^3)$$

Em termos de memória, ao resolver o problema, o único gasto extra e que não é constante está na linha 6, onde é criado um conjunto que pode ter uma quantidade variável

de elementos. No pior caso este conjunto irá conter todos os vértices do grafo e , portanto, a complexidade espacial do algoritmo é dada por:

$$t(n) = O(V)$$

4. Estratégia gulosa

Para resolver o problema por estratégia gulosa, sacrificou-se a otimalidade da solução em favor da complexidade do algoritmo. A estratégia gulosa desenvolvida pode, portanto, não apresentar a solução ótima do problema, mas está limitada por ela através de um fator de aproximação, sendo que a chave principal para o funcionamento da solução se baseia na contagem de conflitos entre os caminhos selecionados. Um conflito acontece toda vez que um foco coberto por um vértice selecionado já estiver sido coberto por um dos vértices do caminho que está sendo analisado. Mais especificamente, seja a função conflito $P(v_i, p)$ definida para $v_i \in V$. Então

$$P(v_i, p) = P(p) + |v_i \setminus (v \in p)|$$

A primeira etapa da solução consiste em calcular todos os caminhos de tamanho mínimo necessário para formar uma solução viável e esta tarefa é realizada com complexidade $\Omega(V)$. Entre estes caminhos, são selecionados aqueles que possuem o menor número de conflitos, formando um universo U de candidatos à solução.

```

1  GULOSO-CANDIDATOS (G , F)
2      let V = G.V
3      let U = empty set
4
5      for each v in V
6          let path = {v}
7
8          for i = 2 to |F|/2
9              path = path + {next v in V}
10
11         U = U + path
12
13     return min{P(U) }
```

A rotina **GULOSO-CANDIDATOS (G , F)** recebe como parâmetros um grafo G e um universo F e constrói todos os caminhos de tamanho mínimo e que possuam a menor quantidade de conflitos. Nota-se que até esta etapa, a rotina executa uma parte da solução por força bruta apresentada anteriormente, mas apenas uma determinada família de caminhos é considerada. É possível perceber ainda que, se os caminhos gerados são mínimos, a solução ótima deverá possuir um tamanho pelo menos igual ou maior ao tamanho dos caminhos mínimos e no mínimo a mesma quantidade de conflitos, ou seja: $|S^*| \geq |p_{min}|$ e ainda $P(S^*) \geq P(p_{min})$ onde S^* representa a solução ótima.

A próxima etapa da estratégia gulosa se baseia em incrementar bidireccionalmente determinado caminho mínimo e selecionar o vértice que gera o menor número de conflitos

para a solução final. Isso implica em analisar dois casos separadamente, que ocorrem quando (i) se adiciona um vértice no início do caminho, anterior ao primeiro elemento e (ii) quando se adiciona um vértice ao final do caminho, posterior ao último elemento. Este processo é repetido enquanto o universo F não tiver sido totalmente coberto. Dado um caminho p_i , o caminho aumentado p_{i+1} é dado por:

$$p_{i+1} = p_i + \min\{P(v_{i\text{ anterior}}), P(v_{i\text{ posterior}})\}$$

Ao aplicar o processo para todos os caminhos gerados na primeira etapa, escolhe-se aquele caminho que possui a menor quantidade de conflitos e esta será a solução do problema, como apresentada no algoritmo **GULOSO(G, F)**. Certamente a solução será viável, mas não necessariamente ótima.

```

1  GULOSO (G, F)
2      let V = G.V
3      let U = GULOSO-CANDIDATOS(G, F)
4
5          //Assuma que cada caminho possui as propriedades first e last,
6          //que representam o primeiro e último vértices do caminho,
7          //respectivamente. Assuma ainda que cada vértice possui as
8          //propriedades l e r, com ponteiros para o vértice anterior e
9          //posterior, respectivamente.
10         for each p in U
11             while p does not cover U
12                 let esquerda = p.first.l;
13                 let direita = p.last.r
14
15                 p = p + min{P(esquerda), P(direita)}
16
17         return min{P(U)} //Retorna o caminho com menor conflito.
18

```

A análise de complexidade do algoritmo **GULOSO(G, F)** começa pela linha 10, onde cada caminho gerado na linha 3 precisa ser analisado. No pior caso, a quantidade de caminhos válidos retornados será da ordem $|V|$. Para cada caminho p_i o *loop* da linha 11 poderá ser executado até $|V| - |p_i|$ vezes, e no pior caso $|p_i| = 3$. Sendo assim, a complexidade do algoritmo é dada por:

$$f(n) = V \times (V - 3) = O(V^2)$$

Para a complexidade espacial, a linha 3 precisa armazenar até $|V|$ caminhos na pior hipótese e as linhas subsequentes não fazem adições significativas ao custo de memória. Portanto, a solução gulosa tem um gasto de memória da ordem de:

$$t(n) = O(V)$$

. 5. Resolução por programação dinâmica

A resolução do problema por programação dinâmica certamente foi o maior desafio enfrentado neste estudo dada a dificuldade em se definir a subestrutura ótima dos problemas de menor tamanho. Todavia, percebe-se que é possível percorrer o problema com uma abordagem semelhante àquela proposta na estratégia gulosa.

O algoritmo dinâmico se baseia essencialmente em percorrer cada combinação de caminhos e armazenar, em uma tabela, a quantidade de colisões e o universo formado por aquele caminho. Ao final do problema, quando todas as posições tiverem sido preenchidas, a resposta estará no elemento que possui a menor quantidade de colisões e que forma o universo completo. Esta abordagem não implica em uma ordem específica para o problema principal porque houve uma transformação do problema proposto para um problema mais fácil de se abordar via programação dinâmica.

Na tabela proposta, cada elemento $e[i, j]$ corresponde ao caminho partindo do vértice i e chegando no vértice j , tomados sempre no sentido horário. Seja um caminho $p_{i \rightarrow j}: v_i \rightarrow v_j$. As colisões deste caminho podem ser calculadas considerando as colisões do caminho $p_{i \rightarrow j-1}: v_i \rightarrow v_{j-1} + P(v_j)$, onde $P(v_j)$ corresponde às colisões causadas pela adição do vértice v_j ao caminho $p_{i \rightarrow j-1}$.

Os casos de base deste problema correspondem ao caminho de apenas um único vértice e ao caminho contendo todos os vértices. Um caminho $p_{i \rightarrow j} | i = j$ por definição não possui nenhuma colisão pois trata-se de um único vértice cobrindo o espaço de focos. Já os caminhos contendo todos os vértices necessariamente correspondem ao caso do maior número de colisões possível. Chamando de $T(i, j)$ a solução para um caminho $p_{i \rightarrow j}$, a equação de recorrência proposta é:

$$T(i, j) = \begin{cases} 0, & i = j \\ T(i, j - 1) + P(j), & i < j \\ T(i - 1, j) - P(i - 1), & i > j \end{cases}$$

A equação de recorrência mostra que para se calcular o resultado para um caminho $p: v_i \rightarrow v_j$ é necessário conhecer o resultado de T para um caminho menor, o que implica em uma mudança de processamento por memória, caracterizando um algoritmo de programação dinâmica.

Por outro lado, quando a matriz estiver completa, é necessário percorrer seus elementos em busca da solução ótima S^* , que é caracterizada por: $S^*: (\min\{P(p)\}, U)$. Ou seja, buscamos o elemento que contenha a menor quantidade de colisões e que forme o universo U .

A análise de complexidade deste problema mostra que é necessário criar e percorrer um total de $|V|^2$ caminhos qualquer que seja a entrada, e ainda, ao final do problema, é necessário percorrer a matriz de solução em busca do resultado que satisfaça

a solução ótima. Além disso, em cada etapa, é necessário gerar os conjuntos com um custo cumulativo que pode chegar até $|F|$. Dessa forma, o custo total da solução é:

$$f(n) = F \times V^2$$

Quanto ao consumo de memória, é necessário armazenar uma matriz de tamanho $|V| \times |V|$ que contém elementos que podem ter tamanho até $|F|$. Sendo assim, o custo de espaço é dado por:

$$t(n) = F \times V^2$$

6. Implementação e testes

A implementação dos algoritmos 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.

7. Análise dos resultados

Os testes foram executados para todas as entradas fornecidas, bem como diferentes entradas geradas manualmente para o pior caso do problema. Os resultados estão sumarizados na figura 3 e serão discutidos posteriormente.

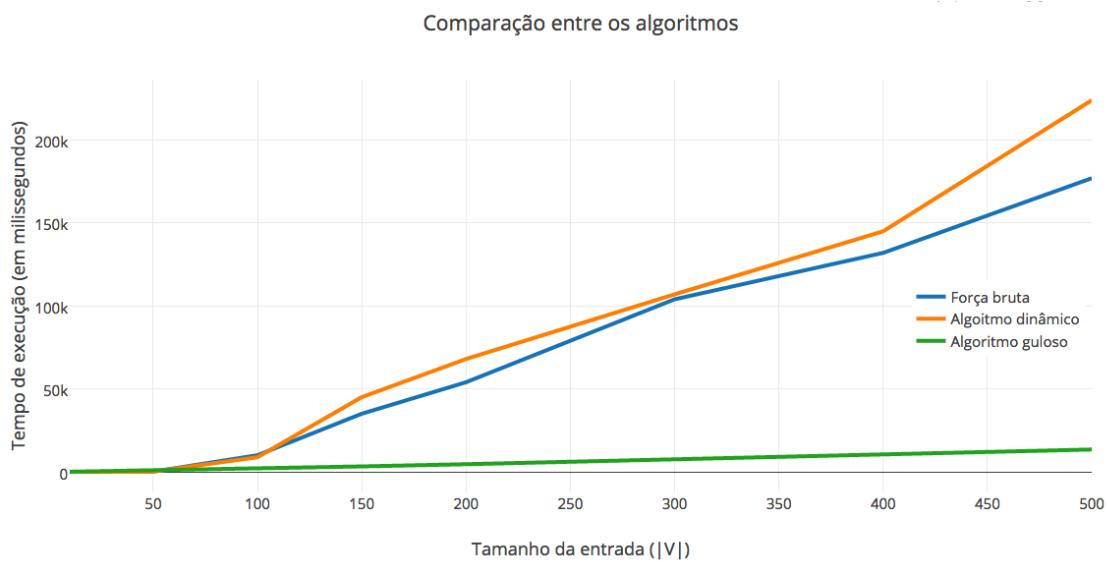


Figura 3. Tempo de execução em ms pelo tamanho da entrada n. O algoritmo dinâmico e força bruta possuem comportamento exponencial, ao passo que o algoritmo guloso se mostrou mais eficiente em todos os casos.

É possível perceber que existe uma equivalência entre os resultados do algoritmo dinâmico e força-bruta. Isso se deve essencialmente ao fato de que, em ambas as abordagens, a solução consiste em analisar sistematicamente todo o espaço de buscas a fim de se encontrar o melhor resultado. A diferença está no fato de que o algoritmo dinâmico realiza esta busca de forma incremental, ao passo que o algoritmo força bruta funciona por tentativa e erro.

Por outro lado, o algoritmo guloso se mostrou mais eficiente em todos os casos, sendo que este ganho em eficiência se equilibra com a perda de otimalidade da solução.

8. 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) A implementação feita em C++ faz uso intenso das bibliotecas STL (*Standard Template Library*), principalmente para a construção de conjuntos e vetores. Embora as soluções desenvolvidas tenham as ordens de complexidade apresentadas, é importante ressaltar que as classes de template `std::set<T>` e `std::multiset<T>` possuem baixa eficiência para as operações de inserção, remoção e comparação de elementos. Uma busca exige pelo menos $O(lgn)$ no pior caso e uma inserção pode utilizar até $O(n)$. Essa baixa eficiência certamente irá implicar em um custo adicional nos algoritmos desenvolvidos, o que justifica o fato de o algoritmo dinâmico ter sido menos eficiente do que o algoritmo força bruta na prática.
- 2) Existem abusos de notação ao longo da modelagem matemática deste trabalho. Como exemplo, considere a expansão do caminho para o algoritmo guloso: $p_{i+1} = p_i + \min\{P(v_{i_anterior}), P(v_{i_posterior})\}$. Neste caso, o valor retornado por $\min\{a, b\}$ corresponde a um número real, quando se espera na realidade um vértice. Nestes casos, suponha que as operações sejam compatíveis.

Referências

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2011). *Introduction to Algorithms*. Cambridge, Massachusetts: MIT Press and McGraw-Hill.
- Ren, W., & Zhao, Q. (2011). A note on ‘Algorithms for connected set cover problem and fault-tolerant connected set cover problem’. *Theoretical Computer Science* , 412 (45), 6451-6554.
- Shuai, T.-P., & Hu, X.-D. *Connected Set Cover Problem and Its Applications*. Beijing University of Post and Telecom, Department of Mathematics.
- Weisstein, E. W. (s.d.). *Binomial Theorem*. Acesso em 30 de 04 de 2016, disponível em Wolfram MathWorld: <http://mathworld.wolfram.com/BinomialTheorem.html>

Zhang, Z., Gao, X., & Wu, W. (2009). Algorithms for connected set cover problem and fault-tolerant connected set cover problem. *Theoretical Computer Science* , 410 (8-10), 812-817.

Trabalho Prático: ZikaZeroAnelDual

José Laerte 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 **ZikaZeroAnelDual**. O problema consiste em determinar, dentre uma corrente de voluntários, uma rede de colaboração coesa que permita combater todos os focos do mosquito transmissor (*Aedes aegypti*). A fim de alcançarmos soluções, foram utilizados três abordagens distintas, que verificam dentre os subconjuntos de voluntários dados, aqueles capazes de combater todos os focos da doença. Dessa forma, este relatório apresenta tais soluções e analisa-as 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, por fim, em conformidade com a especificação do trabalho.

Modelagem do Problema

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 **ZikaZeroAnelDual** propõe a Campanha Corrente do Bem: as pessoas, voluntárias, desafiam-se a formar linhas de frete para acabar com tais criadouros. Nessa corrente, cada voluntário indica um próximo que, em seguida, indica um novo, até que o último indique o primeiro e, assim, cada um possa visitar e eliminar exatamente **dois** criadouros.

O problema, então, consiste em: dado um conjunto V de voluntários associados mutualmente pela relação cíclica 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 existe nenhum voluntário isolado) e (ii) todos os focos em F sejam acessíveis (ou seja, combatíveis).

Dessa maneira, o problema pode ser modelado utilizando um grafo anel: 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 F' de focos $f \in F$ os quais eles podem combater. Essa relação pode ser modelada associando a cada vértice o subconjunto F' de focos acessíveis pelo voluntário v . Por fim, como se trata de um grafo anel, sabemos que o grau de cada vértice é exatamente igual a dois e a cardinalidade de cada subconjunto F' associado a um voluntário também é igual a dois.

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, apresentaremos neste trabalho 3 abordagens de complexidade assintótica polinomial: uma busca por força-bruta; uma segunda utilizando programação dinâmica; e, por último, um algoritmo aproximativo guloso. Suas complexidades teóricas serão analisadas e, então, comparadas com resultados experimentais obtidos a partir de suas implementações na linguagem Java.

Exercício 1. Algoritmos Propostos

Com o objetivo de gerar soluções para o problema **ZikaZeroAnelDual** descrito anteriormente, podemos utilizar diversos paradigmas de programação para modelarmos algoritmos distintos. Nesse sentido, propomos 3 abordagens: uma Força Bruta, uma Programação Dinâmica e um Algoritmo Guloso. Neste Exercício, apresentamos cada um deles.

Busca por Força-Bruta

A fim de encontrarmos solução para o problema apresentado com um algoritmo força bruta, utilizamos o fato de termos um grafo em anel para produzirmos exaustivamente soluções candidatas em tempo polinomial. O Algoritmo 1 apresenta essa solução.

Algorithm 1: GetNetworkBruteForce

```

input : Grafo  $G = (V, A)$ , conjunto  $F$  de focos.
output: Conjunto resposta  $V' \subset V$ .

1 PossibleAnswers  $\leftarrow \emptyset$ 
2 smallerAnswerSize  $\leftarrow \infty$ 

3 foreach  $v \in V$  do
4    $S \leftarrow \emptyset$ 
5    $Next \leftarrow v$ 
6   while reachedFocus( $S$ )  $\neq |F|$  do
7      $S \leftarrow S \cup Next$ 
8      $Next \leftarrow getNext(Next, S)$ 
9   end
10  if  $|S| == smallerAnswerSize$  then
11    PossibleAnswers  $\leftarrow PossibleAnswers \cup S$ 
12  end
13  if  $|S| < smallerAnswerSize$  then
14    PossibleAnswers  $\leftarrow S$ 
15    smallerAnswerSize  $\leftarrow |S|$ 
16  end
17 end
18 return chooseAnswer(PossibleAnswers)

```

Como podemos observar, o Algoritmo avalia, em sentido horário, o tamanho dos caminhos que geram uma solução completa utilizando cada um dos vértices como raiz (Linhas 3-17). Os subconjuntos de menor cardinalidade que satisfazem esses critérios são, então, elegíveis como resultado (Linhas 10-15). Aquele que satisfizer o critério de desempate (menor soma dos ID's) é retornado como resposta (Linha 18).

Para tanto, as funções auxiliares `reachedFocus(S)` e `getNext(Next, S)` trabalham verificando se o caminho S atinge todos os focos e fornecendo o próximo vértice a ser adicionado ao mesmo, respectivamente. Para cada vértice, construímos um caminho possível utilizando tais funções (Linhas 6-9) e, em seguida, adicionamos ao conjunto `PossibleAnswers` aqueles de menor cardinalidade. Por fim, a função `chooseAnswer(PossibleAnswers)` escolhe o resultado, seguindo o critério apresentado.

Programação Dinâmica

Algorithm 2: GetNetworkDynamicProgramming

```

input : Grafo G = (V, A), conjunto F de focos.
output: Conjunto resposta V' ⊂ V.

1 PossibleAnswers ←  $\emptyset$ 
2 Paths[][] ←  $\emptyset$ 
3 Next ←  $v$ 
4  $k \leftarrow 0$ 
5 while Next != null do
6   | Paths[0][ $k$ ] ← {Next}
7   | if reachedFocus(Paths[0][ $k$ ]) == |F| then
8   |   | PossibleAnswers ← PossibleAnswers  $\cup$  Paths[0][ $k$ ]
9   | end
10  | Next ← GetNext(Next)
11 end

12 for  $i=1$  to |V| do
13   | if |PossibleAnswers| > 0 then
14   |   | break
15   | end
16   | for  $j=0$  to |V| do
17   |   | index ←  $j + 1$ 
18   |   | if index == |V| then index ← 0
19   |   | Paths[ $i$ ][ $j$ ] ← Paths[ $i - 1$ ][ $j$ ]  $\cup$  Paths[ $i - 1$ ][index]
20   |   | if reachedFocus(Paths[ $i$ ][ $j$ ]) == |F| then
21   |   |   | PossibleAnswers ← PossibleAnswers  $\cup$  Paths[ $i$ ][ $j$ ]
22   |   | end
23   | end
24 end

25 return chooseAnswer(PossibleAnswers)

```

Uma outra abordagem, utilizando Programação Dinâmica, é apresentada no Algoritmo 2. Nela, utilizamos a matriz $Paths = V \times V$ para computarmos os possíveis caminhos do grafo e testarmos o total de focos que ele acessa. Dessa forma, iniciamos a primeira linha de $Paths$ com caminhos unitários numa ordem de caminhamento total, tal que $Paths[0][k]$ seja conexo de $Paths[0][k+1]$ (Linhas 5-11). Em seguida, descemos os níveis da matriz (Linhas 12-24), aumentando um vértice ao caminho através da união dos caminhos do nível anterior na mesma posição j e $j+1$ (ou 0 para fecharmos o ciclo). A cada formação de um novo caminho, testamos a quantidade de focos acessados e armazenamos soluções candidatas em `PossibleAnswers`.

Uma vez que a construção dos caminhos é feita de forma incremental, quando encontramos uma solução candidata cujo total de focos acessados é máximo (Linha 20), o Algoritmo continua testando apenas os possíveis caminhos de mesma cardinalidade e, então, finaliza. Por fim, a função `chooseAnswer(PossibleAnswers)` escolhe a melhor solução de acordo com o mesmo critério apresentado.

Algoritmo Gulosos

Uma abordagem gulosa para solucionar o problema é apresentada no Algoritmo 3. Nela, iniciamos calculando o **grau de colisão** de cada vértice (Linha 2): total de repetição de focos acessados pelo vértice com relação aos demais. Com esse valor, ordenamos os vértices de forma crescente (Linha 3) e escolhemos o de menor grau para iniciar a busca, adicionando-o na solução S (Linha 5). Escolhemos, então, de forma gulosa, o próximo vértice a ser adicionado ao caminho. Para tanto, utilizando a função `greedyChoiceOfNext(S)` que escolhe, entre os dois vértices adjacentes à S , aquele que agrupa mais focos. Adicionamos este vértice à solução e repetimos o processo até que todos os focos sejam visitados pelos vértices $v \in S$.

Algorithm 3: GetNetworkGreedy

```

input : Grafo G = (V, A), conjunto F de focos.
output: Conjunto resposta V' ⊂ V.

1  $S \leftarrow \emptyset$ 
2  $calculateColision(V)$ 
3  $Next \leftarrow orderByColision(V)[0]$ 
4 while  $reachedFocus(S) \neq |F|$  do
5   |  $S \leftarrow S \cup Next$ 
6   |  $Next \leftarrow greedyChoiceOfNext(S)$ 
7 end
8 return S

```

Como podemos observar, essa abordagem gulosa utiliza uma heurística de escolha do primeiro vértice para iniciar a busca e, então, constrói o caminho solução de forma incremental. Apesar de sempre nos retornar respostas válidas, temos que nem sempre elas serão ótimas, podendo ter cardinalidades maiores que a ótima. Entretanto, como veremos no Exercício 2, tal abordagem apresenta um ganho significativo em termos da ordem de complexidade da solução.

Exercício 2. Análise de Complexidade

No Exercício 1 apresentamos três algoritmos para resolver o problema do **ZikaZeroAnelDual** estudado neste trabalho. Eles baseiam-se em (i) busca exaustiva por uma solução, (ii) programação dinâmica e (iii) algoritmo guloso. Neste Exercício, faremos as análises de complexidade temporal e espacial de cada um deles.

Busca por Força-Bruta

Para fazermos as análises de complexidade do Algoritmo 1, vamos, inicialmente, analisar assintoticamente cada uma das operações mais custosas:

1. `reachedFocus(S)`: A operação de focos alcançáveis soma a quantidade de focos distintos que os vértices do subconjunto $S \subset V$ alcançam. Para tanto, ele percorre cada um dos vértices $v \in S$, acessando o número de focos que ele alcança e somando a uma variável total. No pior caso, o subconjunto S tem cardinalidade igual a $|V|$. Dessa forma, essa operação tem custo temporal de $\mathbf{O}(|V|)$. A complexidade espacial é $\mathbf{O}(1)$ uma vez que apenas uma variável é alocada para armazenar o valor total.
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 subconjuntos $V' \subset V$ da lista `PossibleAnswers` e, para cada um, percorrer seus $v' \in V'$ elementos, somando seus ID' s. Temos que, no pior caso, a lista possui $|V|$ subconjuntos para cada um dos possíveis caminhos iniciando por todos os vértices $v \in V$. Além disso, temos que todos os subconjuntos V' tem cardinalidade de no máximo $|V|$. Portanto, podemos afirmar que a complexidade temporal dessa operação é $\mathbf{O}(|V|^2)$. A complexidade espacial é também $\mathbf{O}(1)$ uma vez que armazenamos apenas o valor da soma dos ID' s de cada subconjunto.

Para descobrir os custos totais do Algoritmo 1, vamos somar todos os custos descritos anteriormente, desconsiderando os custos das demais operações, uma vez que são constantes. Dessa forma, somamos o custo da operação `reachedFocus(S)` executada $|V|^2$ vezes (Linhas 3-17), mais o custo da operação `chooseAnswer(PossibleAnswers)`. As complexidades temporais e espaciais são, respectivamente:

$$\mathcal{O}(|V|^3 + |V|^2)$$

e

$$\mathcal{O}(|V|^2 + 1)$$

Como estamos interessado na complexidade do Algoritmo utilizando notação assintótica, vamos considerar apenas os termos de maior custo do somatório. Portanto, podemos concluir que o custo total temporal e espacial são, respectivamente:

$$\mathcal{O}(|V|^3)$$

e

$$\mathcal{O}(|V|^2)$$

Programação Dinâmica

Para o Algoritmo 2, podemos utilizar os valores dos custos das operações `reachedFocus(S)` e `chooseAnswer(PossibleAnswers)` calculados anteriormente para chegarmos ao seu custo temporal e espacial. Dessa forma, temos que o custo temporal pode ser dado pela soma do custo da operação `reachedFocus(S)` executada $|V|^2$ vezes, mais o custo da operação `chooseAnswer(PossibleAnswers)`. Portanto, o custo temporal total é igual a:

$$O(|V|^3 + |V|^2)$$

Desconsiderando o termo de menor custo, podemos afirmar que a complexidade assintótica temporal do Algoritmo 2 é:

$$O(|V|^3)$$

Para o custo espacial, entretanto, consideramos que o Algoritmo armazena uma matriz de tamanho $|V| \times |V|$, que contém, em cada posição, no máximo $|V|$ elementos. Dessa forma, a complexidade espacial tem um limite superior assintótico dado por:

$$O(|V|^3)$$

Algoritmo Guloso

Por fim, no Algoritmo 3 temos que apenas a operação `reachedFocus(S)` é executada até que o conjunto S conte todos os focos. No pior caso, quando apenas o conjunto V completo seja solução, a operação será executada $|V|$ vezes. Portanto, as complexidades temporais e espaciais são, respectivamente:

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

e

$$O(|V|)$$

Exercício 3. *Implementação*

A fim de testarmos os Algoritmos 1, 2 e 3 com entradas interessantes e fazer as medições experimentais, fizemos uma implementação dos mesmos utilizando a linguagem Java. Por se tratar de uma linguagem Orientada a Objeto, utilizamos algumas Classes para representar estruturas de grafos e resolver as instâncias do problema. Essas estruturas serviram de arcabouço para os Algoritmos e estão descritas brevemente a seguir:

- **ZikaZeroAnelDual**: Classe que resolve as instâncias do problema **ZikaZeroAnelDual**, aplicando o Algoritmos 1, 2 e 3, 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 funções auxiliares.

- **Main:** Classe geral que recebe os dados de entrada em arquivo, no formato especificado no problema, monta o grafo correspondente, e gera a solução para essa instância do **ZikaZeroAne1Dual**, imprimindo o 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.

Exercício 4. Execução Experimental

Caracterização dos Experimentos

Os experimentos descritos neste Exercício 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, dos métodos que implementam os Algoritmos descritos). 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.

Testes de Implementação

Para avaliarmos os Algoritmos 1, 2 e 3, executamos a implementação desenvolvida e apresentada no Exercício 3 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	Força Bruta		Programação Dinâmica		Algoritmo Gulos	
				Tempo	Memória	Tempo	Memória	Tempo	Memória
in0	7	7	6	3	1331	2	1331	2	1331
in1	6	6	4	2	1331	1	1331	1	1331
in2	9	9	8	4	1331	3	1331	2	1331
in3	9	9	8	3	1331	2	1331	2	1331
in4	9	9	8	2	1331	2	1331	2	1331
in5	17	17	9	3	1331	3	1331	4	1331
in6	7	7	6	1	1331	1	1331	2	1331
in7	7	7	2	1	1331	1	1331	2	1331
in8	7	7	14	2	1331	2	1331	2	1331
in9	7	7	8	1	1331	1	1331	2	1331
in10	7	7	6	2	1331	2	1331	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, tais como:

1. Resultado com apenas **um vértice**.
2. Resultado com **todos os vértices**.
3. **Vários possíveis resultados** de mesmo tamanho.

No total, 11 instâncias foram testadas, exercitando aspectos importantes das entradas e dos algoritmos. Para todas elas, obtivemos os resultados esperados para o problema ¹, com exceção de algumas execuções do algoritmo guloso. Isso se dá pelo fato de termos um algoritmo aproximativo que, apesar de sempre retornar resultados válidos, nem sempre resulta em soluções ótimas.

Testes de Desempenho

A análise dos Algoritmos no Exercício 2 mostra que as suas complexidades são proporcionais ao número de vértices do grafo. Dessa forma, a fim de testarmos o desempenho temporal e espacial da nossa implementação, executamos uma sequência de testes com instâncias variando o número n de voluntários e, consequentemente, o valor m de amizade entre eles, mas mantendo fixo o valor de r igual a 5.

A Tabela 2 apresenta os resultados obtidos para cada uma das instâncias, com uma dada variação de voluntários por instância. O tempo é dado em milissegundo e a taxa de utilização de memória em kb.

n	m	r	Força Bruta		Programação Dinâmica		Algoritmo Guloso	
			Tempo	Memória	Tempo	Memória	Tempo	Memória
1	1	5	1	1331	1	1331	0	1331
2	2	5	1	1331	1	1331	1	1331
3	3	5	1	1331	1	1331	1	1331
4	4	5	1	1331	1	1331	0	1331
5	5	5	1	1331	1	1331	1	1331
6	6	5	2	1331	2	1331	1	1331
7	7	5	2	1331	2	1331	1	1331
8	8	5	2	1331	2	1331	1	1331
9	9	5	1	1331	1	1331	1	1331
10	10	5	2	1331	1	1331	1	1331
20	20	5	1	1331	3	1996	0	1996
30	30	5	3	1999	4	1999	1	1999
40	40	5	3	1996	6	1996	2	1996
50	50	5	6	1996	7	1996	2	1996
100	100	5	4	1996	15	2662	1	2662
1000	1000	5	21	7987	147	6610	4	7269
10000	10000	5	63	32635	8753	18274	24	24725
100000	100000	5	1636658	64871	1917	49356	1091	72096

Tabela 2. Testes de Desempenho: variação de n e m, com r = 5.

Como podemos observar, o tempo de execução e o uso de memória crescem de maneira diretamente proporcional ao número de voluntários. A fim de observarmos tal comportamento de maneira intuitiva, os gráficos das Figuras 1, 2 e 3 apresentam os valores obtidos nas execuções descritas anteriormente.

¹ As entradas e saídas testadas estão disponíveis no pacote `data` da solução.

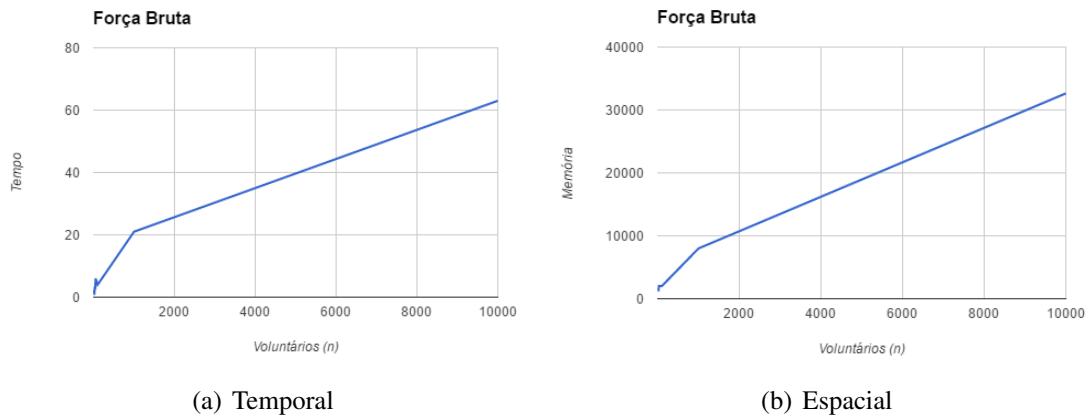


Figura 1. Desempenho do Algoritmo Força Bruta

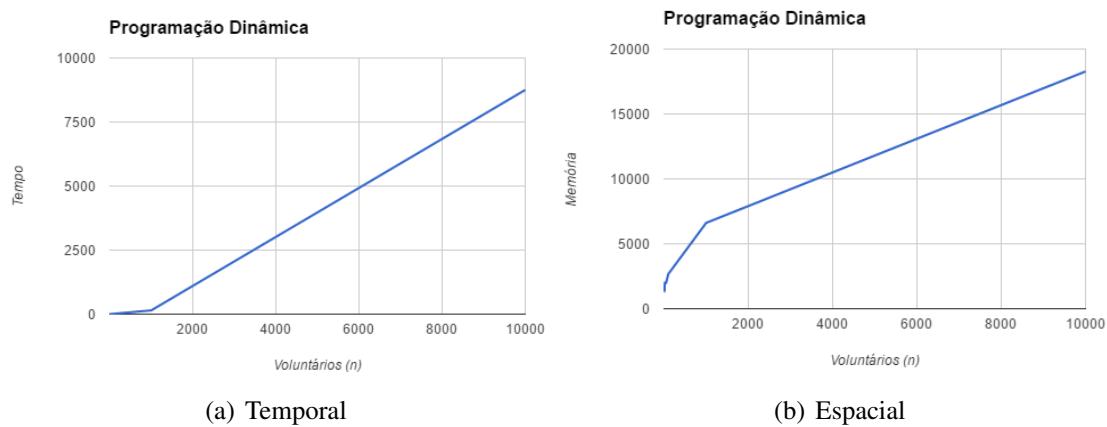


Figura 2. Desempenho do Algoritmo Programação Dinâmica

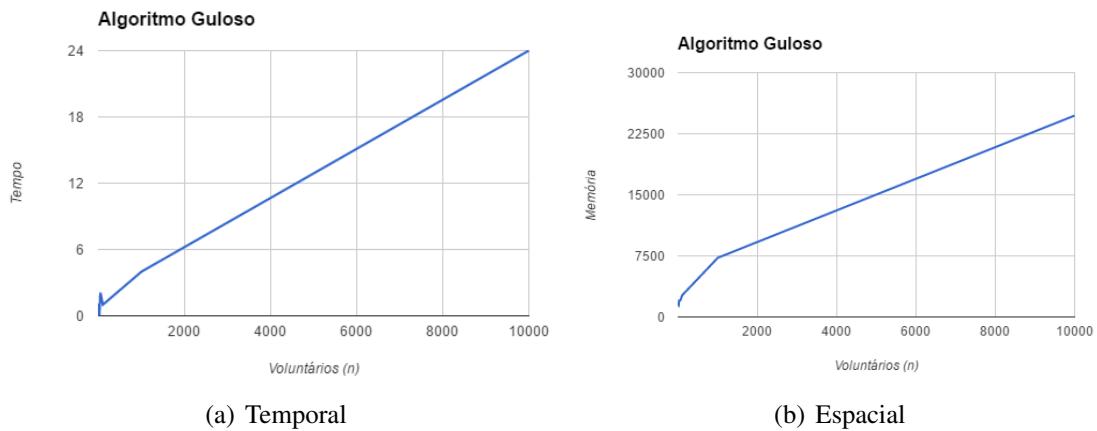


Figura 3. Desempenho do Algoritmo Gúloso

Exercício 5. Discussão

Com os resultados obtidos no Exercício 4, podemos fazer uma última análise acerca das previsões teóricas e os resultados práticos obtidos nas execuções. Para tanto, observamos o comportamento das curvas geradas nas Figuras 1, 2 e 3 para os valores de tempo de execução e utilização de memória.

Como os Algoritmos analisados são da classe de complexidade polinomial, 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 das Figuras 4, 5 e 6 apresentam os valores resultantes das execuções da Tabela 2, comparados a funções polinomiais.

Como podemos observar, os valores obtidos são da mesma ordem de grandeza e ambas as curvas (temporal e espacial) estão, em todos os casos, limitadas superiormente pelas funções derivadas no Exercício 2. Dessa forma, podemos confirmar as nossas previsões teóricas para todos os casos.

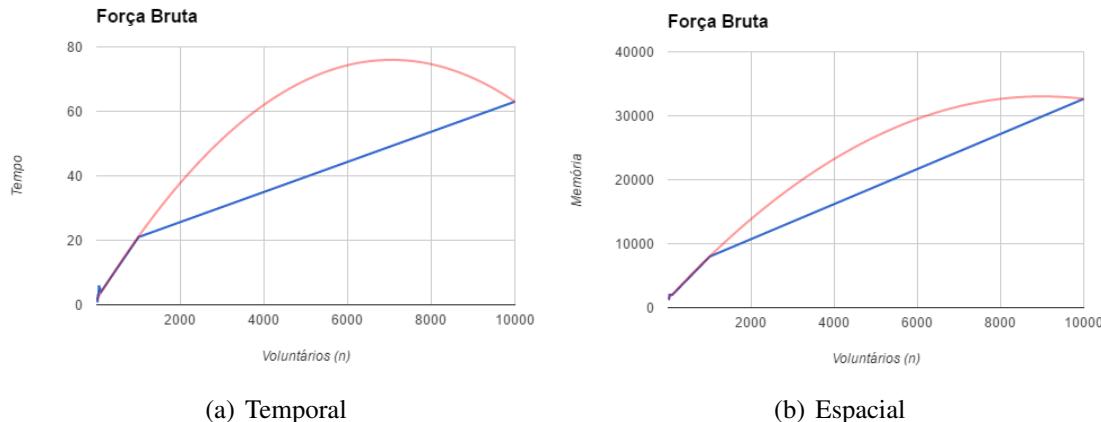


Figura 4. Comparação do Desempenho do Algoritmo Força Bruta

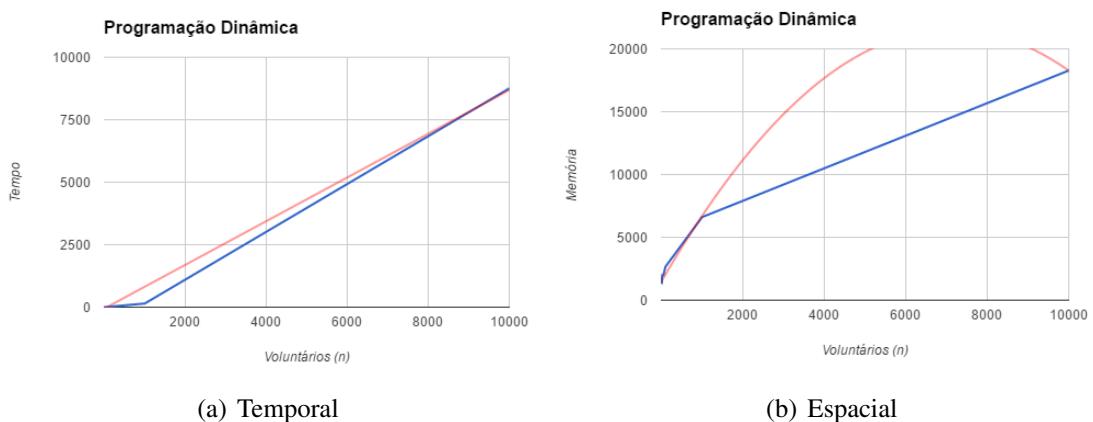


Figura 5. Comparação do Desempenho do Algoritmo Programação Dinâmica

Conclusão

Neste trabalho, analisamos o problema do **ZikaZeroAnelDual**, que consiste em determinar, dentre uma corrente de voluntários, uma rede de colaboração coesa que cubra

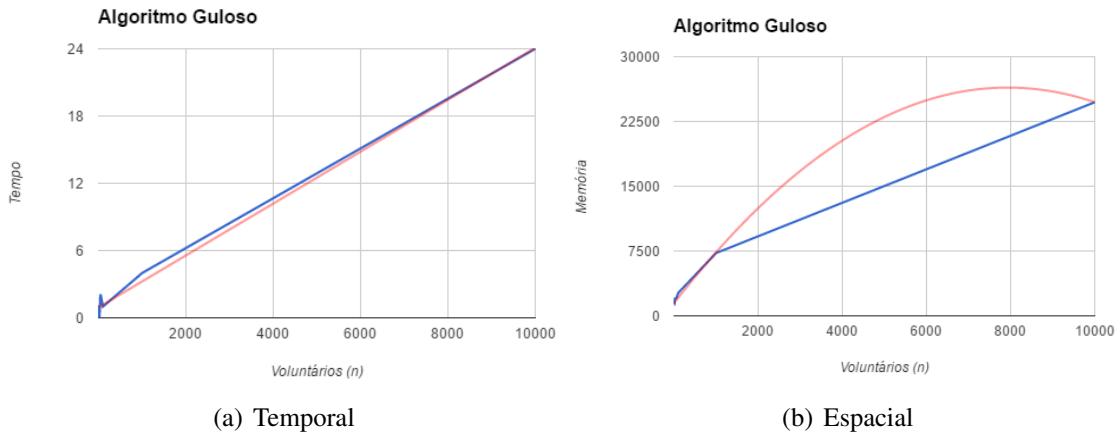


Figura 6. Comparação do Desempenho do Algoritmo Guloso

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, foram descritos e analisados três algoritmos que o solucionem: um por busca exaustiva, outro por programação dinâmica e o último por abordagem gulosa. As análises experimentais foram coletadas a partir de implementações reais do problema utilizando a linguagem Java e os valores obtidos foram comparados com as avaliações teóricas desenvolvidas.

Documentação para o Trabalho de Projeto e Análise de Algoritmos - Paradigmas

Ricardo Barbosa Kloss
UFMG Computer Science Department
Belo Horizonte, Brazil

2016 - 01

1 Modelagem

Nessa seção explicamos as modelagens feitas para resolver o problema do ZikZeroAnelDual. O fato do grafo ser um grafo anel faz com que o problema possa ser resolvido de forma ótima em tempo polinomial, diferente do que acontecia no trabalho de grafos. Nas subseções a seguir explicamos a modelagem para os paradigmas, força-bruta programação dinâmica e algoritmo guloso

1.1 Força Bruta

Na abordagem força bruta consideramos que uma solução viável é definida por um par de nós que em contrapartida definem as extremidades de uma lista no grafo anel. Dessa forma, precisamos olhar os $\theta(n^2)$ pares no grafo e ver quais definem um caminho que atinge todos os focos e desses, qual é o com menor número de vértices. Para cada par precisamos olhar os nós que estão no meio do caminho partindo da extremidade a esquerda andando a direita até a extremidade a direita, esse caminho é no pior caso de tamanho n . Em cada nó verificamos os focos que o mesmo chega, mas, como a quantidade de focos de cada nó é 2, então isso é amortizado. Por fim a **complexidade final** do método é $\theta(n^3) = \theta(n^3)$, sendo n o número de vértices.

Para ilustrar, com 3 vértices nós teríamos os seguintes pares:

$$(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3).$$

Existem pares com nós idênticos pois isso permite explorar soluções de tamanho 1, e existe tanto o par (i, j) quanto o par (j, i) pois isso permite explorar um caminho tanto no sentido horário quanto no anti-horário.

1.2 Guloso

Para o algoritmo guloso, nós escolhemos um nó inicial, uma estimativa para esse nó seria escolher o nó com menor interseção de focos com seus dois vizinhos. Em cada iteração do método escolhemos se incluíremos um nó na extremidade esquerda ou direita da solução existente, nossa escolha é de incluir o nó que mais aumenta a quantidade de focos alcançados.

A complexidade dessa solução seria $O(n)$, pois o custo de verificar qual nó vizinho mais aumenta a solução é definido por 4 comparações, e portanto constante, e a solução não pode ter tamanho maior que n , entretanto, para ter uma maior garantia do ótimo, aplicamos o algoritmo usando cada nó como nó de partida, e assim, a solução passa a ter **complexidade** $O(n^2)$, caso usassemos o nó que tem menor interseção de focos alcançáveis com seus vizinho a complexidade seria $O(n \log(n))$.

Como última observação, em caso de empate na escolha, o algoritmo olha os próximos vizinhos e empatando novamente ele escolhe uma direção aleatória (esquerda ou direita).

1.3 Programação Dinâmica

Como estratégia de programação dinâmica escolhemos resolver a seguinte equação de recorrência:

$$C(n) = \text{opt}_i(K(i, n))$$

Isto é, a solução ótima é a melhor solução do subproblema, $K(i, n)$ que é definido por, o menor caminho começando de i passando por até n nós que alcança a maior quantidade de nós. O subproblema por sua vez é definido pela seguinte equação:

$$K(i, n) = \text{opt}_j(K(i, n - 1) + v_j)$$

Aonde v_j é um elemento no conjunto $\{a, b\} \cup \emptyset$, em que a é o vizinho da extremidade da solução a esquerda e b o vizinho da extremidade da solução a direita, incluímos o vazio, pois assim que todos os focos são alcançáveis, não é necessário incluir mais vértices. Assim, esse subproblema se resolve em no máximo $O(n)$, como isso deve ser feito para $C(n)$, que considera cada nó, a **complexidade final** é $(O(n^2))$, entretanto esperamos que na prática a complexidade pareça menor que isso, pois o primeiro subproblema não precisa olhar n nós, se encontrar uma solução viável já ao explorar uma quantidade j qualquer de nós, com $j < n$.

Além disso, a dependência entre os estados de busca é bem linear, e cada estado depende só do anterior, por exemplo, o menor caminho usando 2 nós começando por i , é composto do menor caminho usando 1 nó começando por i , assim, só precisamos guardar, ou memoizar, a última iteração e o gasto de memória é $O(1)$.

2 Testes experimentais

Nessa seção discutimos os resultados experimentais das diferentes implementações.

Nessa seção apresentamos plots relacionando o tempo gasto com o tamanho de entrada do problema. Para realizar estes testes geramos grafos anéis de forma aleatória, garantindo apenas que existia um conjunto de vértices que alcançavam todos os focos.

2.1 Força Bruta

Na Figura 1c podemos ver que o algoritmo força bruta está equivalente a um crescimento cúbico com a entrada em n , o que era esperado pela teoria. Entretanto esperava que sua curva fosse um pouco mais baixa, acredito que a implementação pudesse ter sido mais eficiente, diminuindo assim o custo de constantes e tendo então uma curva ainda cúbica, mas, mais baixa.

Na Figura 1d temos o tempo pelo número de vértices, aonde o número de focos é constante, nesse teste podemos notar que o resultado é próximo do exposto em 1a, isto porque neste método a complexidade praticamente não depende em nada do número de focos, uma vez que todos os estados viáveis são sempre explorados.

2.2 Algoritmo Guloso

Na Figura 2a podemos ver o gráfico de número de vértices por tempo, podemos notar que a curva ficou bem linear, apesar da complexidade teórica ser quadrática, isto se dá porque o algoritmo não explora mais estados assim que achar uma solução viável para cada nó inicial, e o tamanho da solução tem relação com a quantidade de focos, por exemplo, com 2 focos a menor solução tem tamanho 1 e com $2n$ focos a menor solução usa todos os vértices.

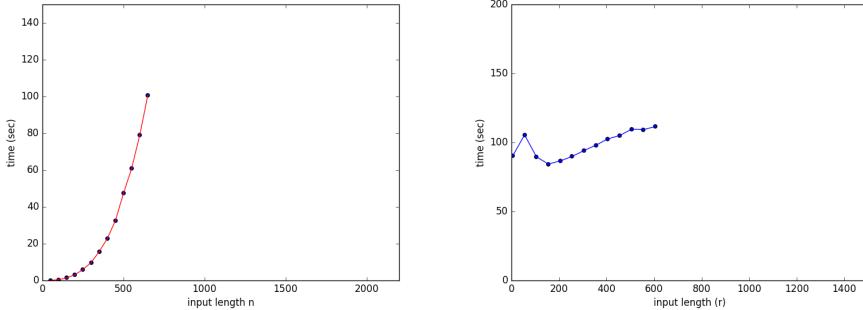
Já na Figura 2b temos um gráfico do número de focos pelo tempo, sendo que o número de vértices permanece constante em 1500. Neste gráfico podemos notar um crescimento acima do linear.

Por fim na Figura 2c temos o gráfico do tempo em função do número de vértices com o número de focos sendo o mesmo do de vértices, e nesse caso a curva fica similar a uma curva quadrática, o que era esperado pela complexidade teórica, isto porque quando a quantidade de focos se aproxima de $2n$ a complexidade do algoritmo se aproxima do seu limite superior que demos como sua complexidade teórica.

2.3 Programação Dinâmica

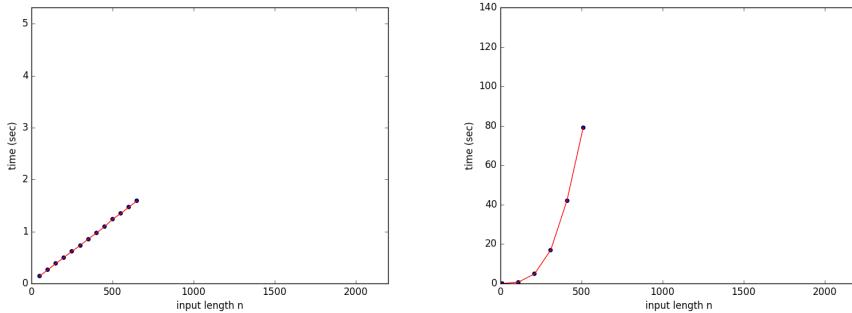
Na Figura 3 temos o sumário dos testes.

Na Figura 3a temos o gráfico do tempo em função do número de vértices com número de focos fixo em 50, podemos perceber um crescimento linear,



(a) Gráfico de tempo por número de vértices do algoritmo força-bruta.

(b) Gráfico de tempo por número de focos do algoritmo força-bruta.



(c) Gráfico de tempo^{1/3} por número de vértices do algoritmo força-bruta. O fato desse gráfico ser linear ($x = y$) indica que o método é muito próximo de uma complexidade cúbica em n .

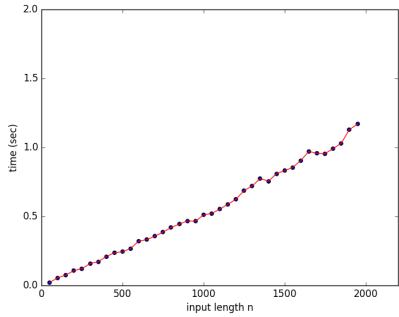
(d) Neste Gráfico o número de focos é igual ao de vértices.

Figura 1: Força Bruta. Para os gráficos (a) e (c), n variou de 50 a 650 com um passo de 50 e o número de focos foi constante em 50.

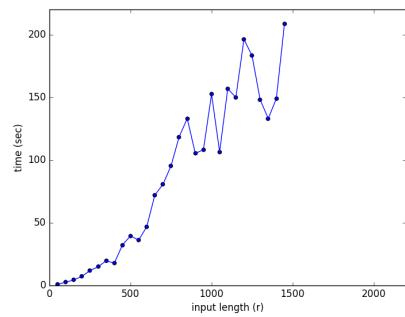
assim como no caso do guloso isso se dá porque o método não olha todos os estados possíveis para descobrir a solução, a não ser que a menor solução tenha tamanho n .

Na Figura 3c nós vemos como se comporta o tempo em função do número de vértices quando o número de foco é o mesmo que o de vértices, nessa situação a complexidade se aproxima do limite superior que foi definido como $O(n^2)$

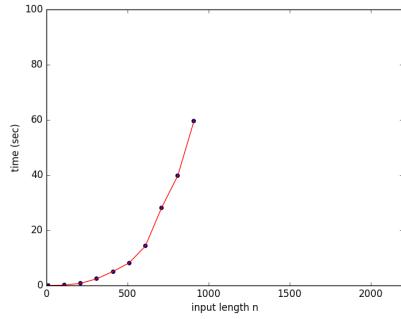
Por fim para mostrar o impacto do número de focos na solução, que é relacionado com o tamanho da solução mínima, na Figura 3b podemos ver que para um mesmo n , igual a 1500, o tempo gasto cresce de forma praticamente quadrática.



(a) Gráfico de tempo por número de vértices do algoritmo guloso.



(b) Gráfico de tempo por número de focos do algoritmo guloso.

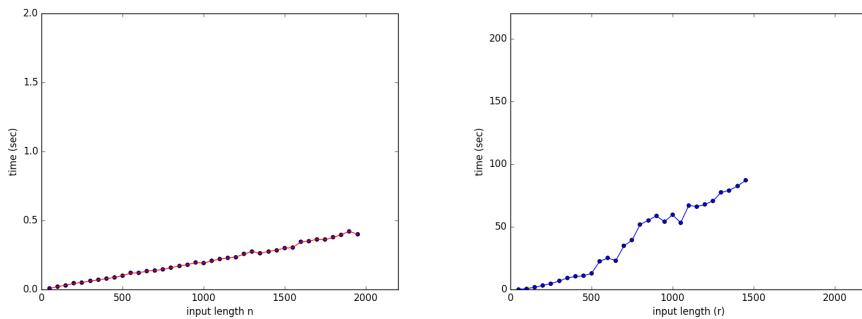


(c) Neste Gráfico o número de focos é igual ao de vértices.

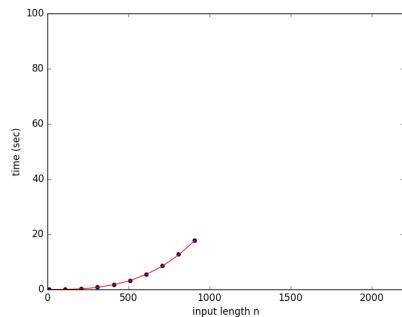
Figura 2: Guloso. Para o gráfico (a), n variou de 50 a 1950 com um passo de 50 e o número de focos foi constante em 50. Para o gráfico (b), n foi fixado em 1500 e f variou como mostrado no gráfico.

3 Conclusão

Neste trabalho desenvolvemos algoritmos polinomiais para resolver o problema ZikaZeroAnelDual, e apesar de termos observados tempos lineares em algoritmos quadráticos concluímos que isso se dá pois o tempo quadrático é o limite superior e em vários casos o algoritmo guloso e programação dinâmica se aproximam do linear.



(a) Gráfico de tempo por número de vértices da programação dinâmica. **(b)** Gráfico de tempo por número de focos da programação dinâmica.



(c) Neste Gráfico o número de focos é igual ao de vértices.

Figura 3: Programação Dinâmica. Para o gráfico (a), n variou de 50 a 1950 com um passo de 50 e o número de focos foi constante em 50. Para o gráfico (b), n permaneceu constante em 1500 e o número de focos variou de

Trabalho Prático 2

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 paradigmas da disciplina de Projeto e Análise de Algoritmos (PAA), da Universidade Federal de Minas Gerais (UFMG), que propõe o problema ZikaZeroAnelDual. Esse problema foi resolvido com a utilização de grafos e paradigmas de programação. Serão apresentadas as soluções desenvolvidas para o problema, assim como suas características, complexidades e resultados.

1. Introdução

Os paradigmas de projeto de algoritmos são uma área da computação que estuda abordagens para a solução de problemas de forma eficiente. Problemas com características similares tendem a ter a mesma forma de resolução. Portanto, um paradigma fornece uma espécie de “template” que auxilia na resolução de determinado problema eficientemente. Existem diversos paradigmas consolidados na literatura, como por exemplo, o Força Bruta, Guloso, e Dinâmico.

Esse trabalho apresenta um problema cuja solução será tratada com os três paradigmas citados anteriormente. Suas características, complexidades e resultados serão discutidos ao longo deste documento.

1.1. Descrição do Problema

Considera-se uma Campanha Corrente do Bem na qual as pessoas desafiam-se a formar linhas de frente para acabar com os criadouros. Inicialmente, há um primeiro voluntário, que indica um amigo, que por sua vez indica outro, e assim sucessivamente, até que o último indica o primeiro (fechando um ciclo). Além disso, nessa campanha, cada voluntário deve visitar exatamente dois criadouros.

Dado um grafo anel (aquele em que todo vértice conecta-se a exatamente outros dois) $G(\mathbb{V}, \mathbb{A})$, no qual \mathbb{V} é o conjunto de n voluntários e \mathbb{A} é o conjunto de $m = |\mathbb{V}|$. laços de amizade, 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, sendo que $R|v| = 2, \forall v \in \mathbb{V}$

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_0 seja conexo. Um exemplo de solução do problema é ilustrado na Figura 1.

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 os vértices adjacentes a v . Também será populada uma lista de focos (Figura

Instância ZikaZeroAnelDual – entrada. .

```
7 7
3 7
7 2
2 5
5 6
6 4
4 1
1 3
6
5 6
1 2
1 3
4 6
2 3
3 5
2 4
```

Instância ZikaZeroAnelDual – saída. .

```
1 3 7
```

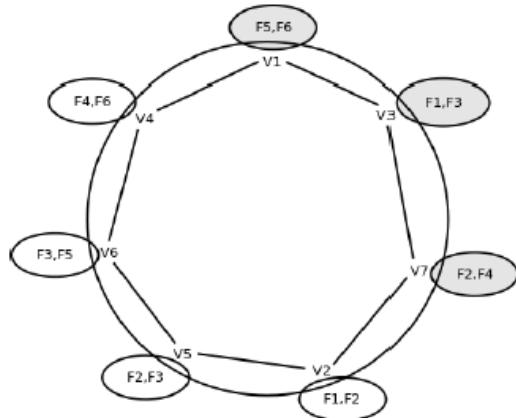


Figura 1. Problema Zika Zero Anel Dual

```
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 2. Formato do arquivo de entrada

4), que para cada posição, referente a um vértice $v \in \mathbb{V}$, armazenará um *set* contendo os dois focos ao qual o vértice tem acesso. O formato do arquivo de entrada deve respeitar às restrições indicadas na Figura 2 e os números contidos em cada linha dele devem ser inteiros.

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 ($\text{Aresta}(u,v) = \text{Aresta}(v,u)$).

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 índice do vértice) e um *value* (adjacências deste vértice).

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

A solução para o problema será ordenada crescentemente, retornada e salva no arquivo de saída (Figura 3).

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

Figura 3. Formato do arquivo de saída

```
$ ./compilar-bruto.sh      #ou
./compilar-dinamico.sh    #ou
./compilar-guloso.sh
```

Figura 4. Comandos para compilação das soluções

2. (Exercício 1) Algoritmos

Dadas as características do problema, serão descritos os pseudocódigos para ilustrar os principais pontos das soluções desenvolvidas.

As soluções devem ser compiladas através da linha de comando correspondente (Figura 4) e executadas através da linha de comando (Figura 5) 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 raiz da aplicação.

2.1. Força Bruta

O paradigma de Força Bruta tem como característica a varredura por todo o espaço de solução à procura da solução ótima. Podem ser feitas “poda” ao longo da varredura para reduzir o número de comparações, já que algumas soluções podem não satisfazer as restrições do problema antes mesmo de serem totalmente descobertas.

Para o caso do problema proposto, visto que cada voluntário acessa exatamente dois focos, o número mínimo de voluntários necessário para acessar determinada quantidade de focos r pode ser dado por $r/2$. Portanto, as podas são realizadas de forma a ignorar possíveis soluções que apresentam quantidade de vértices (voluntários) inferior a este mínimo.

Serão gerados subgrafos conexos dos vértices do grafo, variando p de 1 a n , sendo p a quantidade de elementos do subgrafo a ser gerado 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 será retornado como solução para o problema e salvo no arquivo de saída (Figura 3), pois, como os subgrafos 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.

O Algoritmo 1 ilustra o pseudo-código da abordagem Força Bruta.

Primeiramente é calculada a quantidade mínima de vértices necessária para que se tenha acesso à determinada quantidade de focos r (linha 9). Então, inicia-se a geração das

```
$ ./executar-bruto.sh entrada saida      #ou
./executar-dinamico.sh entrada saida    #ou
./executar-guloso.sh entrada saida
```

Figura 5. Comandos para execução das soluções

Algorithm 1: Pseudo-código da abordagem Força Bruta para o problema ZikaZeroAnelDual

Data: G

Result: Solução do problema ZikaZeroAnelDual

```
1  $min \leftarrow r/2;$ 
2 for  $size \leftarrow min$  to  $v - 2$  do
3   foreach  $v \in \mathbb{V}$  do
4      $coverFocus \leftarrow \emptyset;$ 
5      $extractedSubgraph \leftarrow \emptyset;$ 
6      $lastVertex \leftarrow v;$ 
7      $extractedSubgraph \cup lastVertex;$ 
8      $coverFocus \leftarrow coverFocus \cup focus[lastVertex];$ 
9     for  $size \leftarrow 1$  to  $size - 1$  do
10     $newVertex \leftarrow lastVertex.adjacencias[0];$ 
11    if  $newVertex \in extractedSubgraph$  then
12       $newVertex \leftarrow lastVertex.adjacencias[1];$ 
13    end
14     $extractedSubgraph \cup newVertex;$ 
15     $coverFocus \leftarrow coverFocus \cup focus[newVertex];$ 
16     $lastVertex = newVertex;$ 
17  end
18  if  $|coverFocus| = r$  then
19     $sort(extractedSubgraph);$ 
20    return  $extractedSubgraph;$ 
21  end
22 end
23 end
24  $coverFocus \leftarrow \emptyset;$ 
25  $extractedSubgraph \leftarrow \emptyset;$ 
26 foreach  $v \in \mathbb{V}$  do
27    $coverFocus \leftarrow focus[v];$ 
28 end
29 if  $|coverFocus| = r$  then
30    $sort(extractedSubgraph);$ 
31   return  $extractedSubgraph;$ 
32 end
33 return  $\emptyset;$ 
```

possíveis soluções (subgrafos), partindo da quantidade mínima de vértices necessários. Gera-se subgrafos conexos de G de tamanho $size$ (linha 9), partindo de cada $v \in \mathbb{V}$ (linha 3). Para cada subgrafo gerado, verifica-se se ele cobre todos os focos (linha 18), caso sim, o subgrafo é ordenado e retornado como solução para o problema (linha 20). Caso nenhum subgrafo de tamanho $size$ solucione o problema, gera-se subgrafos de tamanho $size + 1$ até $n - 1$ ou até a que solução seja encontrada.

Se nenhum subgrafo de tamanho até $n - 1$ tiver acesso a todos os focos, computa-se os focos acessados pelo conjunto \mathbb{V} (linha 26) e testa-se se ele soluciona o problema (linha 29). Caso sim, ele será ordenado e retornado como solução para o problema (linha 31), caso contrário, a solução é um conjunto vazio (linha 33).

Como o paradigma Força Bruta varre todo o espaço de solução e seleciona a melhor, ele sempre vai encontrar a solução ótima, caso ela exista, apesar do elevado custo necessário para encontrá-la.

2.2. Programação Dinâmica

O paradigma de programação dinâmica 'quebra' o problema a ser resolvido em subproblemas mais simples e combina a solução desses subproblemas para alcançar a solução final.

O algoritmo desenvolvido utiliza uma estrutura auxiliar S , um mapa no qual cada *key* representa o primeiro vértice de determinada sequência de vértices que formam uma possível solução. Cada *key* possui como *value* um *set* de focos aos quais esta solução acessa.

Foi preenchida uma lista *seq* na qual o grafo foi representado de forma 'direcionada', de forma a evitar ambiguidade. Para isso, fez-se um 'corte' no grafo, na posição do primeiro vértice, então tem-se uma sequência única de adjacentes à determinado vértice. Essa transformação não altera o problema, pois todos os possíveis subgrafos originais continuam a existir.

Cada posição do mapa representa um dos n vértices do grafo original e a linha que representará a solução, ao final da execução, possuirá r linhas.

Uma vez que os focos acessados em determinado subgrafo $\{v_1, \dots, v_{size}\}$ foram computados, para verificar os focos acessados por $\{v_1, \dots, v_{size}, v_{size+1}\}$ basta apenas adicionar os focos acessados por v_{size+1} ao mapa criado.

A solução dinâmica pode ser representada pela equação de recorrência 1.

$$[S[i]] = \begin{cases} \text{focus}[\text{seq}(i: v_{size})], & se S[i] = \emptyset \\ S[i] \cup \text{focus}[\text{seq}(i + size)] & \text{caso contrário} \end{cases} \quad (1)$$

O Algoritmo ilustra o pseudo-código da abordagem Programação Dinâmica.

2.2.1. Sobreposição de subproblemas

Ao buscar o conjunto de voluntários com tamanho n , no primeiro passo temos à disposição todo o conjunto V' de voluntários. Se adicionarmos o voluntário v_{size} à

Algorithm 2: Pseudo-código da abordagem de Programação Dinâmica para o problema ZikaZeroAnelDual

Data: G

Result: Solução do problema ZikaZeroAnelDual

```
1  $min \leftarrow r/2;$ 
2  $subgraphAux \leftarrow \emptyset;$ 
3  $lastVertex \leftarrow v[0];$ 
4  $subgraphAux \cup lastVertex;$ 
5 foreach  $v \in \mathbb{V}$  do
6    $newVertex \leftarrow lastVertex.adjacencias[0];$ 
7   if  $newVertex = lastVertex$  or ( $newVertex \in extractedSubgraph$ )
8     then
9        $newVertex \leftarrow lastVertex.adjacencias[1];$ 
10      end
11       $subgraph \cup lastVertex;$ 
12       $lastVertex \leftarrow newVertex;$ 
13    end
14   $S \cup lastVertex;$ 
15  for  $size \leftarrow 1$  to  $n - 2$  do
16     $lastIndex \leftarrow size - 1;$ 
17    foreach  $v \in \mathbb{V}$  do
18      if  $lastIndex = countVertices$  then
19         $lastIndex \leftarrow 0;$ 
20      end
21       $S[i] \cup focus[lastIndex];$ 
22      if  $size \geq min|S[i]| = r$  then
23         $sort(extractedSubgraph);$ 
24        return  $extractedSubgraph;$ 
25      end
26       $lastIndex \leftarrow lastIndex + 1;$ 
27    end
28   $S[0] \cup focus[v - 1];$ 
29  if  $|S[0]| = r$  then
30     $sort(subgraphAux);$ 
31    return  $subgraphAux;$ 
32  end
33 return  $\emptyset;$ 
```

solução, temos que computar os focos acessados por V' e adicionar os focos acessados por v_n . Se no segundo passo adicionarmos o voluntário v_{size+1} , para uma solução de tamanho $(size + 1)$, temos que calcular os focos acessados por $V' \cup v_{size}$ e adicionar os focos acessados por v_{size+1} , e assim por diante, até que a solução do problema seja encontrada. Podemos notar que nessa estrutura os subproblemas estão sobrepostos, ou seja, para resolver um problema precisamos resolver seus subproblemas.

2.3. Gulosos

Foi utilizada uma heurística para solução do problema utilizando o paradigma gulosos.

O paradigma gulosos possui a característica de sempre fazer escolhas ótimas locais, na esperança de alcançar a escolha ótima global. Ou seja, ele seleciona o candidato mais atraente para atingir a solução ótima, sem se preocupar com as consequências dessa escolha para os próximos passos.

Para a solução do problema ZikaZeroAnelDual, é selecionado primeiro o vértice (voluntário) que possui os focos com a menor quantidade de conflitos. Ou seja, computase quantas vezes os focos em cada vértice são acessados por outros vértices. O vértice com os focos que somam a menor quantidade de conflitos será o primeiro adicionado à solução.

Após a seleção do primeiro vértice, adiciona-se novos vértices um a um, escolhendo-se sempre o vértice adjacente à solução que possui a menor quantidade de conflitos com o grafo e que acrescente algum novo foco ainda não acessado pela solução (se possível). Caso a quantidade de vértices do subgrafo gerado seja maior ou igual ao mínimo necessário, ele é verificado e retornado, caso solucione o problema.

Escolheu-se tal abordagem tendo-se em vista que vértices com focos de menor conflito indicam focos raros, com maior probabilidade de aparecerem na solução ótima.

O Pseudo-código 3 ilustra a abordagem Gulosa.

2.3.1. Optimalidade da Solução

Essa abordagem nem sempre vai encontrar a solução ótima. Para que isso aconteça, dependerá dos focos acessados por cada adjacência do primeiro vértice.

Suponha a entrada 6(a) a solução retornada será a solução ótima $V' = \{1, 3, 4\}$. Mas, considerando-se uma entrada 6(b) a solução retornada pela abordagem gulosa será a solução $V' = \{1, 3, 4, 7\}$ e não a solução ótima $V' = \{1, 3, 7\}$.

3. (Exercício 2) Análise das Complexidades Temporais e Espaciais

Para a análise da complexidade das soluções, 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

Como a complexidade da leitura do grafo é assintoticamente menor que a da computação da solução. Considerou-se, portanto, a complexidade da geração das soluções para a análise.

Algorithm 3: Pseudo-código da abordagem Gulosa para o problema ZikaZeroAnelDual

Data: G

Result: Solução do problema ZikaZeroAnelDual

```
1  $min \leftarrow r/2;$ 
2  $subgraph \leftarrow \emptyset;$ 
3  $count \leftarrow \emptyset;$ 
4 if  $min = 1$  and  $|focus[1]| = r$  then
5   | return 1;
6 end
7 if  $n < min$  then
8   | return  $\emptyset$ ;
9 end
10 foreach  $v \in \mathbb{V}$  do
11   | foreach  $v' \in \mathbb{V}$  do
12     |   | if  $v' \neq v$  then
13       |   |   |  $count[v] \leftarrow conflicts[focus[v]]$ ;
14     |   | end
15   | end
16 end
17  $shortest \leftarrow count[0]$ ,  $firstVertex \leftarrow 0$ ;
18 for  $i \leftarrow 1$  to  $n - 1$  do
19   | if  $count[i] < shortest$  then
20     |   |  $shortest \leftarrow count[i]$ ;
21     |   |  $firstVertex \leftarrow i$ ;
22   | end
23 end
24  $coverFocus \leftarrow \emptyset$ ;
25  $lastVertex \leftarrow firstVertex$ ;
26  $subgraph \leftarrow subgraph \cup firstVertex$ ;
27  $coverFocus \leftarrow coverFocus \cup focus[firstVertex]$ ;
28 for  $size \leftarrow 2$  to  $n$  do
29   | if ( $count[adj1] > count[adj2]$  and  $adj1.melhoraSolucao$ ) or
    |   |  $(!adj2.melhoraSolucao)$  then
30     |   |   |  $subgraph \leftarrow subgraph \cup adj1$ ;
31     |   |   |  $firtVertex \leftarrow adj1$ ;
32     |   |   |  $coverFocus \leftarrow coverFocus \cup focus[firtVertex]$ ;
33   | else
34     |   |   |  $subgraph \leftarrow subgraph \cup adj2$ ;
35     |   |   |  $lastVertex \leftarrow adj2$ ;
36     |   |   |  $coverFocus \leftarrow coverFocus \cup focus[lastVertex]$ ;
37   | end
38   | if  $size \geq min$  and  $|coverFocus| = r$  then
39     |   | return  $subgraph$ ;
40   | end
41 end
42 return  $\emptyset$ ;
```

$G(V, A)$

$$n = 7 \quad m = 14 \quad r = 6$$

$$A = \{\{1,3\}, \{3,4\}, \{4,2\}, \{2,5\}, \{5,6\}, \{6,7\}\}$$

$$F = \{\{5,6\}, \{2,3\}, \{1,3\}, \{2,4\}, \{2,3\}, \{3,5\}, \{4,6\}\} \quad F = \{\{5,6\}, \{2,3\}, \{1,3\}, \{4,6\}, \{2,3\}, \{3,5\}, \{2,4\}\}$$

(a) Exemplo de entrada 1

$G(V, A)$

$$n = 7 \quad m = 14 \quad r = 6$$

$$A = \{\{1,3\}, \{3,7\}, \{7,2\}, \{2,5\}, \{5,6\}, \{6,4\}\}$$

(b) Exemplo de entrada 2

Figura 6. Exemplos de entradas

3.1.1. Força Bruta

No melhor caso, o primeiro subgrafo gerado possui acesso à todos os vértices. Para esse caso, a complexidade é $O(r/2 + r/2)$, referente ao custo de se adicionar os $r/2$ vértices à solução e percorrê-los para verificar se eles cobrem todos os focos. A complexidade temporal do melhor caso pode ser representada pela equação 2.

$$O(r) \quad (2)$$

No pior caso da solução proposta, todos os subgrafos de todos os tamanhos serão gerados. Serão gerados subgrafos de n tamanhos ($O(n - r/2)$), para cada tamanho será selecionado cada um dos vértices como vértice inicial ($O(n)$) e para cada um deles será gerado um subgrafo de tamanho $n - 1$ ($O(n - 1)$). Depois será verificado se o grafo atende à solução ($O(n)$). Caso haja solução, ela será ordenada ($O(nlogn)$).

Portanto, a complexidade temporal do pior caso pode ser representada pela equação 3.

$$O((n - r/2) * n * (n - 1) + n + nlogn) \quad (3)$$

Por razões de simplificação, a complexidade temporal do pior caso será representada pela equação 4.

$$O(n^3) \quad (4)$$

Conclui-se, então, que a solução proposta tem complexidade temporal de ordem polinomial em função de n .

3.1.2. Programação Dinâmica

Para uma quantidade de vértices n , a solução dinâmica o algoritmo executa duas estruturas de repetição para gerar as soluções, resultando, no pior caso, em uma complexidade temporal de $O(n^2)$.

Para montar a sequência de vértices é demandada complexidade $O(n)$ e para verificar se a possível solução cobre todos os vértices, apenas se compara r com a quantidade de focos presentes no mapa em $S[i]$, o mesmo é feito em $O(1)$.

A ordenação do resultado é feita apenas uma vez, em $O(n \log n)$.

$$O(n^2 + n + n \log n) \quad (5)$$

Por razões de simplificação, a complexidade temporal do pior caso será representada pela equação 6.

$$O(n^2) \quad (6)$$

3.1.3. Guloso

Para uma quantidade de vértices n , a solução gulosa percorre cada vértice e o compara com os outros $n - 1$ vértices, computando a quantidade de conflitos de cada um. Após isso, percorre-se o vetor que armazena a quantidade de conflitos de cada um dos n vértices, em vista de verificar o vértice que possui a menor quantidade de conflitos.

Após a adição do primeiro vértice, são adicionados os adjacentes a solução com o menor conflito com ela. Esta fase pode ser executada, no pior caso, $(n - 1)$ vezes.

Portanto, a complexidade temporal do pior caso da solução pode ser representada pela equação 7.

$$O(n * (n - 1) + (n - 1) + (n - 1) + n \log n) \quad (7)$$

Por razões de simplificação, a complexidade temporal do pior caso será representada pela equação 8.

$$O(n^2) \quad (8)$$

No melhor caso, o problema pode ser solucionado com apenas um vértice (2 focos), sendo tal solução trivial. A complexidade do melhor caso pode ser representada pela equação

$$O(1) \quad (9)$$

Conclui-se, portanto, que a solução gulosa proposta tem complexidade temporal de ordem polinomial em função de n .

3.2. Complexidade Espacial

Visto que para a solução foi utilizado um mapa de adjacências e um *set* de focos, e que cada vértice possui duas adjacências e acesso a dois focos, considera-se que mapa de adjacências exija memória de $O(n * 2)$, e a lista de focos $O(n * 2)$.

3.2.1. Força Bruta

Para a complexidade de espaço, o algoritmo Força Bruta irá armazenar um conjunto de vértices de tamanho p . Então, sua complexidade no pior caso será de $O(n)$, onde o algoritmo irá armazenar uma quantidade de vértices igual a n na solução. O melhor caso é quando a solução necessita de apenas um vértice, complexidade $O(1)$.

3.2.2. Programação Dinâmica

A estrutura seq possui tamanho n . No mapa, a maior linha possui r focos, possuindo, no pior caso, complexidade espacial $O(nr)$. Estima-se portanto uma complexidade espacial $O(nr + n)$.

3.2.3. Guloso

Para a complexidade de espaço, a solução Gulosa irá armazenar um conjunto de vértices até que alcance a solução. Então, sua complexidade no pior caso será de $O(n)$, onde o algoritmo irá armazenar um número de vértices igual a n na solução. O melhor caso é quando a solução necessita de apenas um vértice, complexidade $O(1)$.

4. (Exercício 4) Testes da Implementação

Para análise da complexidade de tempo, foram realizados experimentos visando verificar o tempo de execução em função do aumento do número de vértices e focos ($r = n$).

Os resultados do tempo de execução, em milissegundos, podem ser observados no gráfico da figura 7 e na tabela 1.

Foram geradas instâncias do problema onde $r = n$ e nas quais só se encontra uma solução com $(n - 1)$.

Tabela 1. Resultado dos experimentos de Tempo

n	Força Bruta	Guloso	Dinâmico
10	4	1	2
20	15	2	4
40	30	3	8
60	69	10	12
120	261	14	47
250	1857	19	110
500	4617	54	129

5. Exercício 5 - Comparação de Análise e Execução

Foi previsto que a complexidade temporal da solução gulosa seria menor que a da solução força bruta ($O(n^2)$) e ainda polinomial. É possível observar no comportamento temporal desta solução no gráfico 7 que há concordância das previsões teóricas com os resultados



Figura 7. Testes de Tempo - Pior Caso

experimentais. Porém, pôde-se verificar que nem sempre o resultado desta solução é a menor quantidade de voluntários, como especificado no problema, apesar de retornar sempre uma solução válida.

Também é possível observar um crescimento de tempo da solução dinâmica, superior ao da solução gulosa e inferior à força bruta, em concordância com a complexidade temporal $O(n^2)$ prevista. Esta solução, entretanto, apresenta maior complexidade espacial se comparada com as outras soluções.

De acordo com a previsão teórica descrita na seção 3, a complexidade temporal da solução força bruta poderia ser expressa por $O(n^3)$, uma classe polinomial diferente do algoritmo de força bruta e do guloso, portanto esperava-se uma maior lentidão do algoritmo força bruta em comparação às outras soluções, o que pôde ser verificado nos resultados do gráfico 7, mostrando concordância das previsões teóricas com os resultados experimentais.

6. Conclusão

Esse trabalho apresentou três paradigmas diferentes para a resolução do problema Zika-ZeroAnelDual. Com isso foi possível perceber a importância do projeto de algoritmos para resolver um problema de maneira eficiente, assim como mostrar o impacto de cada abordagem de acordo com a entrada do problema. Um paradigma eficiente pode requerer mais memória em relação a outro. A escolha de um paradigma ou outro vai depender das características do problema e do cenário de desenvolvimento.

Durante os experimentos, ao variar a quantidade de vértices do grafo original, podemos verificar que o tempo de execução da abordagem de força bruta aumenta em proporções muito maiores, se comparada com as outras.

A abordagem gulosa, apesar de possuir uma complexidade menor em relação ao força bruta, nem sempre consegue retornar a solução ótima para o problema proposto, sendo dependente dos valores do conjunto de entrada.

O paradigma de programação dinâmica, mostra a importância de se identificar a subestrutura ótima e a sobreposição de subproblemas para garantir a otimalidade da solução e reduzir sua complexidade temporal, ao evitar que um subproblema seja computado repetidas vezes ao longo da resolução do problema. Neste paradigma, prioriza-se a redução da complexidade temporal em detrimento da espacial.

Devidos às particularidades do problema, foi possível realizar podas no processo de computação das soluções, o que corroborou para uma redução considerável na complexidade dos algoritmos.

Trabalho prático - Zika Zero Anel Dual

Camila Laranjeira da Silva¹

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

mila.laranjeira@gmail.com

1. Introdução

Este documento é referente ao trabalho prático do módulo *Paradigmas de Programação* da disciplina de Projeto e Análise de Algoritmos, como parte do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais.

2. Exercício 1

2.1. Força Bruta

Este trabalho pode ser considerado um caso especial do problema *Set Cover*, o qual em sua forma original é conhecidamente NP-Completo. No entanto, por se tratar de um grafo anel, e pelo requisito de uma solução conexa, temos limitações suficientes para reduzir drasticamente o número de combinações possíveis, propondo então uma solução força bruta polinomial.

O grafo $G(V, A)$ foi modelado como um grafo direcionado, cujas arestas apontam na direção dos amigos que foram indicados (e.g. para a aresta (u, v) , o voluntário u indicou o amigo v). No caso específico do grafo anel, cada vértice tem exatamente uma aresta partindo dele e outra chegando.

Tendo em mãos a lista de adjacência dessa modelagem, o loop principal da abordagem consiste em realizar buscas por profundidade (*depth-first search - DFS*) para diferentes profundidades máximas d , iniciadas em cada um dos n vértices, como apresentado no algoritmo 1. Para cada conjunto de vértices visitados, é testada a cobertura de criadouros através da função *testCover(visited)*. A primeira solução viável encontrada pode ser retornada, interrompendo as iterações, já que a profundidade d é iterada de forma crescente, ou seja, qualquer solução encontrada no futuro terá tamanho $t \geq d$.

```
begin
    min_depth = r/2;
    for d = min_depth to n do
        for each v ∈ V do
            visited = dfs(G, v, d);
            if testCover(visited) then
                return visited;
            end
        end
    end
end
```

Algorithm 1: Solução proposta dentro do paradigma força bruta.

O problema nos permite estabelecer a restrição de profundidade mínima (\min_depth) para reduzir a quantidade de iterações, pois cada voluntário acessa exatamente dois criadouros, ou seja, a quantidade mínima possível de voluntários para atender os r criadouros é $r/2$.

Os algoritmos 2 e 3 apresentam respectivamente as implementações da busca por profundidade e da função *testCover* utilizadas na solução. Quanto à implementação da *DFS*, vale reforçar a particularidade da modelagem, de modo que cada vértice tem apenas um elemento na sua lista de adjacência, não sendo necessário um loop para iterar nessa lista.

```

Input:  $G(V, A)$ , source,  $d$ ;
begin
    visited = [];
    stack = [source];
    while stack is not empty do
        v = stack.pop();
        if v not in visited then
            visited.add(v);
            if visited.size() ==  $d$  then
                return visited;
            end
            stack.push(adj[v])
        end
    end
end
```

Algorithm 2: Implementação da busca por profundidade

```

Input:  $V'$  (subset de  $V$ );
Output: retorna True caso o subset cubra todos os criadouros, ou False caso contrário;
begin
    covered = zeros(r);
    sum = 0;
    for each  $v \in V'$  do
        for each  $s \in set\_cover[v]$  do
            if not covered[s] then
                covered[s] = 1;
                sum += 1;
                if sum ==  $r$  then
                    return True;
                end
            end
        end
    end
    return False;
end
```

Algorithm 3: Implementação da função para testar cobertura de conjunto.

2.2. Programação Dinâmica

Na solução aqui proposta foi mantida a modelagem em grafos utilizada para a abordagem de força bruta pois, como veremos mais a frente, a geração do conjunto Universo é feita de forma similar à exploração por força bruta.

Primeiramente vamos entender a construção da tabela que irá armazenar as soluções do subproblema. Vamos chamá-la de $T[V][U]$, onde V é o já conhecido conjunto de vértices, e U é o conjunto Universo. As linhas da tabela variam de $1..n$, fazendo referências aos n vértices de V , já as colunas são geradas a partir de combinações de vértices. Pensando de forma simples, a geração desse Universo teria um custo exponencial (gerando um conjunto similar a $\{\{1\}, \{2\}, \dots, \{1, 2\}, \{1, 3\}, \dots, \{1, 2, 3, \dots, n\}\}$), no entanto como já explicado no início da seção 2.1, as limitações do problema proposto possibilitam um custo polinomial, considerando apenas as combinações de vértices conexas.

Vamos considerar o exemplo de entrada simples ilustrado na figura 1. A tabela $T[V][U]$ resultante desse grafo é apresentada na figura 2. A título de ilustração ela foi preenchida de forma completa, incluindo todas as possíveis combinações do grafo anel, no entanto na implementação aqui proposta, o conjunto Universo é preenchido apenas com combinações que gerem coberturas de criadouros ainda não existentes na tabela (e.g. a última coluna não seria incluída, pois a cobertura $\{1, 2, 3, 4\}$ já havia sido gerada por $V3 \cup V1$).

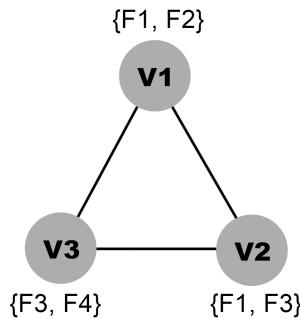


Figura 1. Modelagem em grafos de uma entrada exemplo.

	\emptyset	$V1\{1,2\}$	$V2\{1,3\}$	$V3\{3,4\}$	$V1 \cup V2 \{1,2,3\}$	$V2 \cup V3 \{1,3,4\}$	$V3 \cup V1 \{1,2,3,4\}$	$V1 \cup V2 \cup V3 \{1,2,3,4\}$
0	0	∞	∞	∞	∞	∞	∞	∞
$V1\{1,2\}$								
$V2\{1,3\}$								
$V3\{3,4\}$								

Figura 2. Tabela resultante do grafo da figura 1 para solucionar o problema usando o paradigma de programação dinâmica.

Cada elemento $T[i][X]$, iterando em ambos V e U , contém o mínimo número de elementos de $V' = \{V1, V2, \dots, Vi\}$, capaz de cobrir X , para todo $X \subseteq U$. Seguindo tais regras, o campo $T[n][F]$ (lembrando que $F = \{1, 2, \dots, r\}$ é o conjunto de focos) irá conter a solução procurada. Para preencher a tabela, seguimos a seguinte recorrência:

$$\begin{cases} T[0][\emptyset] = 0 \\ T[0][X \neq \emptyset] = \infty \\ T[i][X] = \min(1 + T[i-1][X \setminus Vi], T[i-1][X]) \end{cases}$$

2.3. Algoritmo Guloso

Para a abordagem gulosa, a modelagem de $G(V, A)$ foi realizada na forma de um grafo não direcionado, ou seja, a lista de adjacência de cada voluntário v contém ambos o amigo indicado por v , e o amigo que v indicou. Tal modelagem é importante para a solução aqui proposta, onde a otimalidade local leva em conta toda a região adjacente de cada vértice, em ambas as direções do grafo anel.

As abordagens gulosas para o problema do *set cover*, costumam ser iniciadas es-colhendo o vértice que cobre o maior número de elementos do conjunto Universo. Como o nosso problema limita cada vértice a uma cobertura exatamente igual a 2, não é possível transpor diretamente tal regra. Por conta disso foi definido um critério de escolha do melhor par para inicializar a solução, o qual foi implementado na função *chooseFirstPair()* (algoritmo 4). O critério segue uma lógica muito parecida com o algoritmo de força bruta, no entanto é feita apenas uma iteração de profundidade $d = 2$, já que estamos procurando o melhor par. A busca termina quando encontra o primeiro par capaz de cobrir 4 criadouros, pois esta é a cobertura máxima possível por um par de voluntários.

```

begin
     $d = 2;$ 
     $\text{max\_cover} = 4;$ 
     $\text{temp\_cover} = [];$ 
    for each  $v \in V$  do
         $\text{visited} = \text{dfs}(G, v, d);$ 
         $\text{cover} = \text{set\_cover}[\text{visited}[0]] \cup \text{set\_cover}[\text{visited}[1]];$ 
        if  $\text{cover.size()} > \text{temp\_cover.size()}$  then
             $\text{temp\_cover} = \text{cover};$ 
             $\text{best\_pair} = \text{visited};$ 
            if  $\text{cover.size()} == \text{max\_cover}$  then
                return  $\text{best\_pair};$ 
            end
        end
    end
    return  $\text{best\_pair};$ 
end

```

Algorithm 4: Implementação da função *ChooseFirstPair*

Tendo em mãos o par que inicializa a solução, o loop principal da abordagem gulosa realiza os seguintes passos:

1. Cria nova lista de adjacência, transformando o par de vértices em um único vértice. O segundo elemento do par deixa de existir e as arestas são reconectadas para o primeiro.
2. Cria nova relação de cobertura de criadouros, unindo os criadouros cobertos por ambos os vértices do par e atribuindo-os ao primeiro elemento do par.

3. Escolhe o melhor vizinho do novo vértice criado, formando um novo par. Essa escolha segue uma lógica simples: o vértice adjacente capaz de cobrir a maior quantidade de criadouros ainda não cobertos, é selecionado.

A cada iteração, os vizinhos ótimos escolhidos são adicionados ao conjunto solução, sendo a iteração interrompida quando tal conjunto atende os requisitos de cobertura de criadouros, cuja implementação foi apresentada no algoritmo 3. O pseudocódigo da solução gulosa aqui descrita se encontra no algoritmo 5.

```

begin
    pair = chooseFirstPair();
    solution = pair;
    new_adj = new_set_cover = [];
    while not testCover(solution) do
        new_adj = getNewAdjacency(adj, pair);
        new_set_cover = getNewSetCover(set_cover, pair);
        pair[1] = chooseBestNeighbour(new_adj, new_set_cover, pair[0]);
        solution.add(pair[1]);
    end
    return solution;
end

```

Algorithm 5: Solução proposta dentro do paradigma guloso.

3. Exercício 2

3.1. Força Bruta

3.1.1. Complexidade Temporal

- Função principal: inicialmente desconsiderando as chamadas de método dentro da função principal, temos dois loops encadeados, um deles iterando entre $[r/2..n]$, e o outro $[1..n]$. A partir disso podemos partir de uma complexidade $O((n - r/2)n) = O(n^2)$
- *DFS*: A primeira operação realizada a cada iteração é uma busca por profundidade. Conhecidamente sabemos que sua complexidade temporal é $O(n+m)$, mas considerando que estamos lidando com um grafo anel, podemos simplesmente atribuir uma complexidade $O(n)$ para essa operação.
- *testCover*: Essa validação também possui dois loops encadeados, tendo o loop externo uma variação $[1..n']$, sendo n' o tamanho do subset sendo avaliado. O loop interno no entanto tem variação fixa de duas iterações, já que o problema traz a limitação de dois criadouros por voluntário. Podemos dizer então que a complexidade desse método vale $O(n)$.

Unindo as ideias, temos a função principal, de custo $O(n^2)$, realizando duas funções de custo $O(n)$, nos permitindo concluir que a complexidade total da solução força bruta é $O(n^3)$

3.1.2. Complexidade Espacial

Para descrever a complexidade espacial do algoritmo, podemos listar as principais estruturas utilizadas:

- Lista de adjacência: Conhecidamente essa estrutura possui complexidade espacial de $\Theta(n + m)$, a qual no caso específico do grafo anel, pode ser considerada $\Theta(n)$
- Cobertura de criadouros: A estrutura que ao longo desse documento foi chamada de *set_cover*, relaciona cada um dos n vértices com dois criadouros, ou seja, sua complexidade espacial é também $\Theta(n)$.
- Solução: a cada iteração do algoritmo, o vetor solução assume valores que variam entre $[1..n]$, podemos definir sua complexidade como $O(n)$.

3.2. Programação Dinâmica

3.2.1. Complexidade Temporal

- Geração do conjunto Universo: O conjunto universo é criado através de uma exploração similar à realizada na função principal da abordagem de força bruta, tendo também uma complexidade polinomial igual a $O(n^2)$.
- Preenchimento da Tabela: A tabela principal $T[V][U]$ tem dimensão $n \times n^2$, de modo que o seu preenchimento possui complexidade temporal $O(n^3)$.

Como a complexidade de um problema é dominada pelo polinômio de maior grau, temos que a abordagem de programação dinâmica tem complexidade final de $O(n^3)$.

3.2.2. Complexidade Espacial

- Lista de adjacência: $\Theta(n)$, como já descrito na subseção 3.1.2.
- Cobertura de criadouros: $\Theta(n)$, como já descrito na subseção 3.1.2.
- Tabela $T[V][U]$: Devido às suas dimensões $n \times n^2$, sua complexidade espacial é $O(n^3)$.

3.3. Algoritmo Guloso

3.3.1. Complexidade Temporal

- Função principal: Tendo apenas um loop *While*, no pior caso a função irá explorar todos os vértices, um a um, tendo complexidade $O(n)$.
- *chooseFirstPair*: No pior caso o método vai iterar entre todas as arestas do grafo, e se tratando de um grafo anel, podemos considerar uma complexidade $O(n)$.
- *getNewAdjacency*: Custo fixo independente do tamanho da entrada, pois apenas reconecta pares de vértices. Portanto: $O(1)$
- *getNewSetCover*: Também $O(1)$ pois lida com um par de vértices, assim como a função *getNewAdjacency*.
- *chooseBestNeighbour*: Também $O(1)$ pois apenas avalia ambos os vizinhos da esquerda e da direita do subgrafo que compõe a solução sendo construída.

A operação *chooseFirstPair* é realizada apenas uma vez, em seguida é iniciado o loop principal, de custo $O(n)$, o qual realiza três operações de custo fixo independente do tamanho da entrada. Podemos afirmar então que a complexidade temporal é $O(n)$.

3.3.2. Complexidade Espacial

- Lista de adjacência: $\Theta(n)$, como já descrito na subseção 3.1.2.
- Cobertura de criadouros: $\Theta(n)$, como já descrito na subseção 3.1.2.
- Estrutura auxiliar de adjacência: Considerando que a solução se baseia em remover o grafo a cada iteração (como já explicado na seção 2.3), temos uma estrutura adicional de complexidade similar à lista de adjacência original, sendo $O(n)$.
- Estrutura auxiliar de cobertura: Pela mesma razão da estrutura auxiliar de adjacência, também temos a réplica da estrutura que regista a cobertura de criadouros, de complexidade espacial também $O(n)$.
- Solução: $O(n)$, como já descrito na subseção 3.1.2.

4. Exercício 3

O arquivo *ZikaZeroAnelDual.zip* enviado junto a esse documento contém o código fonte das soluções implementadas, bem como os scripts para sua compilação e execução. **Note que não há implementação do paradigma de programação dinâmica.**

5. Exercício 4

Nesse exercício será apresentada uma análise da execução das soluções, relacionando diferentes tamanhos de entrada com o impacto no tempo de execução. Para automatização dos testes foi criado um módulo responsável por construir entradas de diferentes tamanhos, variando os valores de n e r , obedecendo os critérios do problema proposto. Vale ressaltar que o problema não permite alterar o valor de m de forma independente, estando ele diretamente relacionado ao valor de n . É importante destacar também que todas as entradas automaticamente geradas tinham uma solução viável com no máximo r voluntários, além disso o valor de n era sempre maior que r , ou seja, não há possibilidade do paradigma de força bruta cair no pior caso. Já no caso do guloso, não é possível fazer tal afirmação.

A primeira bateria de testes foi realizada variando a quantidade de voluntários (n) num intervalo de $[100..1000]$ a um passo de incremento $i = 100$, com quantidade de focos fixada em $r = 50$. Para cada valor de n eram realizados 10 testes, e tirada a média dos tempos de execução. Na figura 3 é possível ver um resultado comparativo entre as abordagens gulosa e força bruta em gráficos de mesma escala.

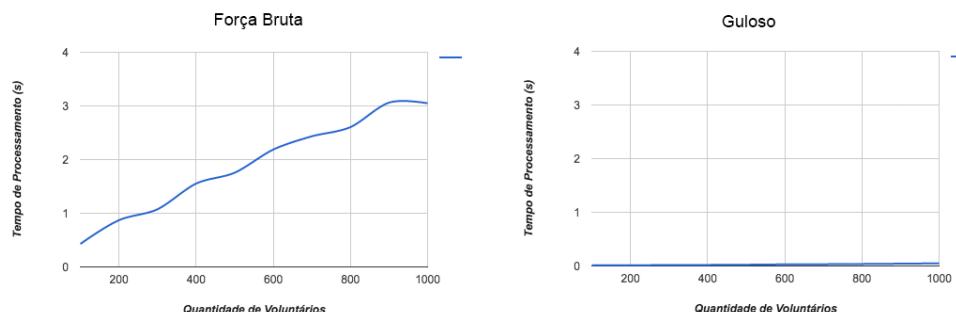


Figura 3. Tempo de processamento (força bruta e gulosa), variando n

É possível perceber a abordagem força bruta cresce muito mais rápido que a gulosa, ao ponto que, quando postas na mesma escala, o crescimento do guloso é quase imperceptível. Por conta disso, a figura 4 traz uma adequação de escala para o segundo paradigma, de modo a facilitar a visualização do comportamento da função. Pode-se perceber que foi necessário reduzir a escala em duas casas decimais para tirar informações relevantes acerca do seu crescimento.

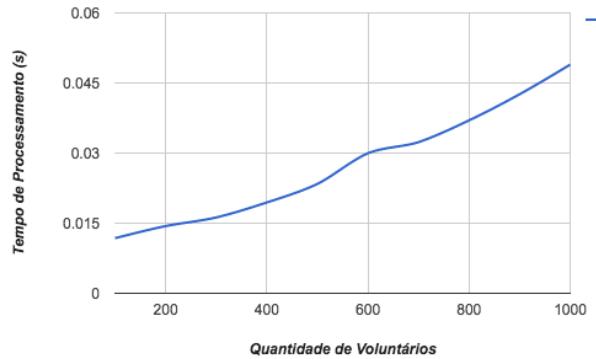


Figura 4. Tempo de processamento da abordagem gulosa, variando n , com escala reduzida para melhor visualização

A segunda bateria de testes fixou a quantidade de voluntários em $n = 200$, dessa vez variando r dentro de um intervalo $[10..100]$ a um passo de incremento $i = 10$. A figura 5 traz o resultado desses testes. Como já dito anteriormente, cada ponto do gráfico é uma média de 10 testes realizados com os mesmos parâmetros.

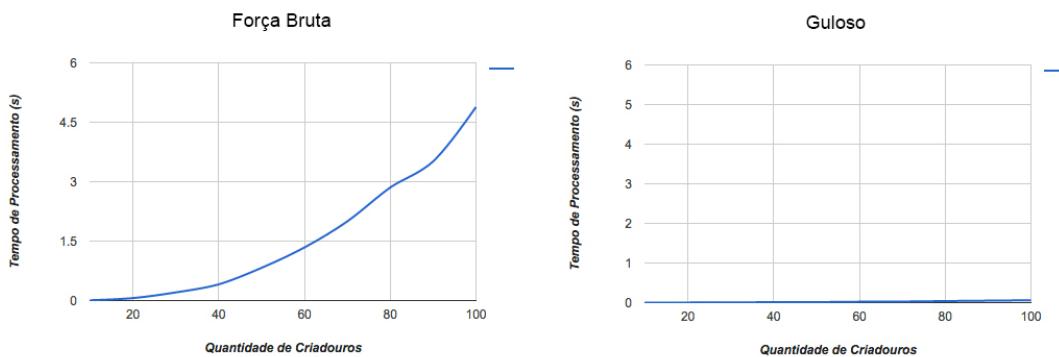


Figura 5. Tempo de processamento (força bruta e gulosa), variando r

Mais uma vez o crescimento da abordagem gulosa é imperceptível em um gráfico cuja escala é adequada para os dados da força bruta. Por isso a imagem 6 novamente traz uma adequação de escala para melhor visualização.

Por fim, o último conjunto de testes foi realizado com entradas sem solução viável, ou seja, não existia conjunto de vértices capaz de cobrir todos os criadouros. Para isso, a geração automática de entradas foi alterada, fazendo com que um dos criadouros fosse

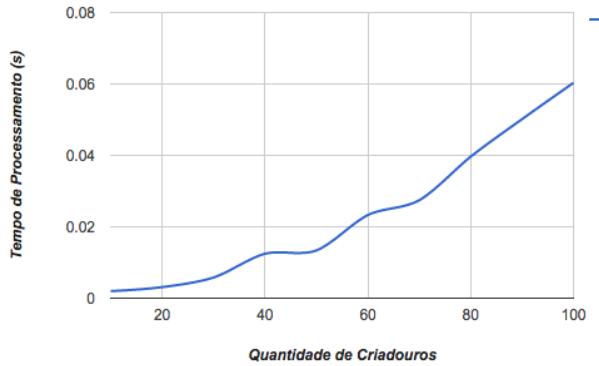


Figura 6. Tempo de processamento da abordagem gulosa, variando r , com escala reduzida para melhor visualização

deixado de fora, não sendo atribuído a nenhum voluntário. O objetivo desse teste é demonstrar o crescimento da função no pior caso de cada abordagem.

Dessa vez começaremos avaliando os tempos de execução para diferentes valores de r , variando entre $[10..100]$ a um passo $i = 10$. O valor fixo de n , no entanto, foi reduzido pela metade em relação à última bateria de testes, sendo definido $n = 100$. Essa redução foi feita para viabilizar o uso da mesma metodologia de testes, consumindo uma quantidade razoavelmente baixa de tempo, mas, como vemos na figura 7, esses parâmetros ainda permitem avaliar o comportamento da função de forma relevante, o qual será melhor discutido na seção 6. Nesse caso foram mantidas as escalas adequadas para cada conjunto de dados.

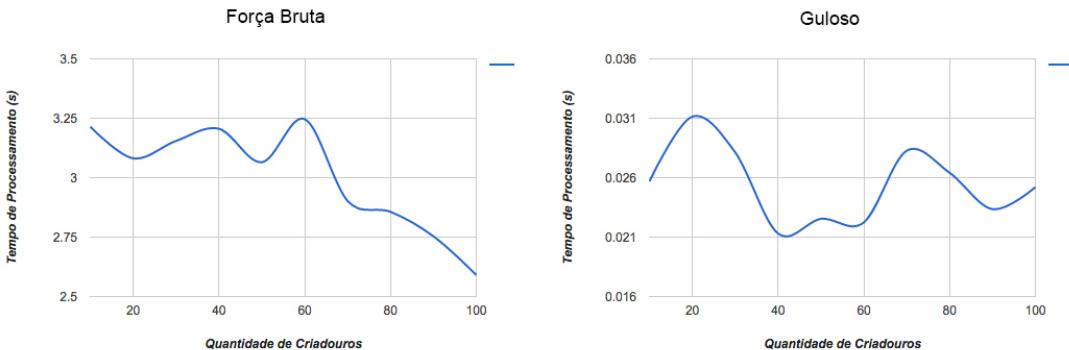


Figura 7. Tempo de processamento (força bruta e gulosa), variando r em entradas sem solução

Ainda tratando do conjunto de testes sem solução, foi realizada uma bateria de testes com $r = 20$, variando $n = [20, 200]$ a um passo de incremento $i = 20$. Os valores de n , bem como o passo de incremento foram reduzidos novamente buscando realizar os testes em um tempo viável. O resultado comparativo entre as abordagens força bruta e gulosa (em escalas adequadas para cada conjunto de dados) é apresentado na figura 8.

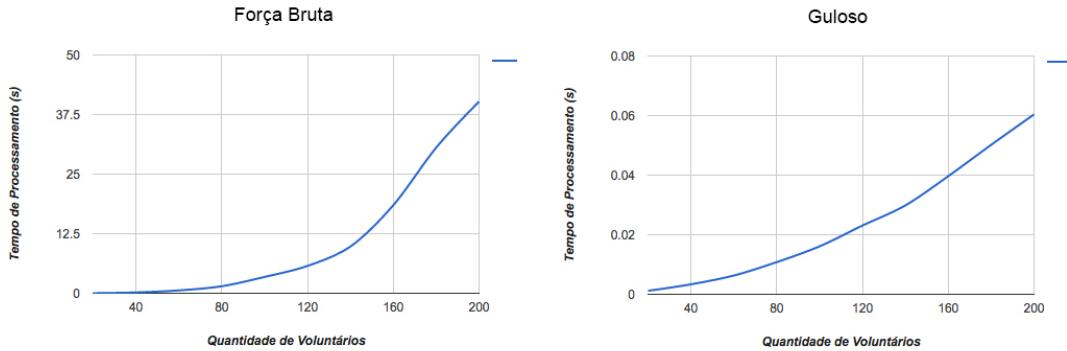


Figura 8. Tempo de processamento (força bruta e gulosa), variando n em entradas sem solução

6. Exercício 5

O primeiro ponto que merece destaque é a evidente discrepância nos tempos de execução das abordagens força bruta e gulosa, tendo a primeira um crescimento muito mais acelerado que a segunda em todos os testes realizados. Tal comportamento era esperado, pois como definido na seção 3, todas as operações da força bruta são muito mais custosas que as gulosas.

Gostaria de chamar atenção especial para os gráficos da figura 7, onde o tempo de execução da abordagem força bruta **decrece** à medida que o r aumenta. Este também é um comportamento previsível, considerando a análise realizada na seção 3, onde foi dito que a complexidade da função principal (sem considerar o custo dos métodos ali chamados) é de $O((n - r/2)n) = O(n^2)$. Isso significa que, quanto maior o número de focos em relação ao número de voluntários, menor é a complexidade real da solução. Nessa mesma figura, temos um comportamento praticamente constante da abordagem gulosa (com pequenas flutuações), independente da quantidade de focos. Isso se dá pois não há solução para a entrada, então a execução não é interrompida até ter explorado todo o grafo.

Já quando a entrada tem solução (lembrando que a geração das entradas garante uma solução de tamanho máximo r), a execução de ambas as abordagens é interrompida ao encontrar a primeira solução viável. No caso da força bruta, cuja exploração itera no intervalo $[r/2..n]$, seu loop principal vai iterar no máximo $r/2$ vezes, justificando o crescimento mais acelerado quando a quantidade de criadouros cresce (figura 5), em relação ao crescimento de n (figura 3). De forma similar, a abordagem gulosa também é interrompida quando encontra a primeira solução viável, por isso o crescimento de r (que define uma solução mínima de tamanho $r/2$) também implica em um aumento no tempo de execução, justificando o crescimento mais acelerado apresentado na figura 6, onde r é variado, em relação à figura 4, onde n é variado.

Na figura 8 podemos visualizar melhor a complexidade de ambas as abordagens, se assemelhando mais ao que foi descrito na seção 3, pois o fato das entradas não terem solução viável, faz com que o algoritmo acabe caindo no pior caso, avaliando todas as possibilidades permitidas pelo grafo, antes de descobrir que não há solução.

ZicaZeroAnelDual

Trabalho Prático 2 - 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 ZicaZeroAnelDual, 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

- Grafo anel: o fato de \mathbb{V} representar um grafo anel permitiu utilizar apenas um vetor para representá-lo. Não é necessário que, dentro do arquivo de entrada, os vértices estejam ordenados por ordem de ligação, pois o programa se encarrega de criar o grafo na ordem correta.
- Matriz de cobertura: mostra quais focos estão sendo cobertos por quais voluntários. Um valor k no elemento ij indica que o j -ésimo voluntário cobre o foco k . O fato de que $|i| = 2$ será utilizado em todos os paradigmas.
- Vetor de focos cobertos: indica quais focos são cobertos por pelo menos um dos voluntários presentes na combinação que está sendo testada.
- Obs: os critérios de desempate diferem entre cada um dos paradigmas.

3. Algoritmos (exercício 1)

O programa começa lendo o valor de n , utilizado para determinar o tamanho do vetor que representa o grafo anel, e de m , utilizado para determinar quantas relações de amizade devem ser lidas (nesse caso deste tipo de grafo, n sempre é igual a m). As m linhas seguintes são então lidas e o grafo é criado.

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 dois índices, correspondendo aos focos aos quais cada um dos voluntários tem acesso. Será com base nesses índices que a matriz de cobertura será preenchida.

3.1. Busca por Força-bruta

A solução proposta realiza uma busca exaustiva no espaço de soluções, envolvendo o teste de cada uma das possíveis combinações existentes para o conjunto \mathbb{V} de voluntários. O Algoritmo 1 mostra o que foi feito para implementar essa abordagem. Para cada posição inicial, uma janela de tamanho crescente percorre o grafo anel. Verifica-se, dentro de uma janela, se ela cobre todos os focos. A primeira janela a conseguir fazer isso é então retornada.

Algorithm 1: ZicaZeroAnelDual - Força bruta

Input: \mathbb{V} , matrizCobertura
Output: \mathbb{V}'

```
1  $\mathbb{V}' = [];$ 
2 for  $tamJanela \leftarrow 1$  to  $\mathbb{V}.tamanho$  do
3   for  $posInicio \leftarrow 1$  to  $\mathbb{V}.tamanho$  do
4      $focosCobertos = [False, False, \dots, False];$ 
5     for  $i \leftarrow posInicio$  to  $posInicio + tamJanela$  do
6        $posAtual = i \bmod \mathbb{V}.tamanho;$ 
7        $focosCobertos[matrizCobertura[\mathbb{V}[posAtual]][0]] = True;$ 
8        $focosCobertos[matrizCobertura[\mathbb{V}[posAtual]][1]] = True;$ 
9
10      if todas as posições do vetor de cobertura estão marcadas como True then
11        for  $i \leftarrow posInicio$  to  $posInicio + tamJanela$  do
12           $\mathbb{V}' = \mathbb{V}' + \mathbb{V}[i \bmod \mathbb{V}.tamanho];$ 
13
14      return  $\mathbb{V}';$ 
```

3.2. Programação Dinâmica

A ideia por trás da implementação dinâmica feita foi a seguinte: criou-se uma matriz de vetores de cobertura. Um elemento ij representa uma combinação que começa no i -ésimo voluntário do grafo anel e termina no j -ésimo voluntário. Na primeira parte do algoritmo, preenche-se essa matriz tendo-se em mente dois casos: (i) i igual a j : o número de focos cobertos é obtido a partir da matriz de cobertura e (ii) i diferente de j : o número de focos cobertos é obtido somando-se o valor obtido a partir da matriz de cobertura com o valor do elemento à esquerda. Para garantir a corretude da matriz, o preenchimento começa pelas diagonais.

Neste ponto a matriz está completamente preenchida. O próximo passo é procurar pela combinação que cobre todos os focos e requer o menor número de voluntários. Na busca a matriz será percorrida diagonalmente, começando pela diagonal principal. O Algoritmo 2 resumo a abordagem implementada.

3.3. Algoritmo Guloso

A ideia por trás da implementação feita para a abordagem gulosa é bastante simples. Seleciona-se um dos voluntários e, a cada iteração, analisa-se, entre os voluntários à sua esquerda e à sua direita, qual dos dois permite cobrir um número mais de novos focos. Para melhorar as soluções geradas, todos os voluntários são selecionados como voluntários iniciais em um determinado momento. A solução final é então a melhor entre todas as obtidas. O Algoritmo 3 mostra a sequência de passos realizados durante o processo.

O algoritmo não é ótimo, mas produziu uma solução ótima em 34 de 35 testes que foram feitos (em instâncias fornecidas aos alunos em conjunto com outras utilizadas nos testes experimentais).

4. Análise de Complexidade (exercício 2)

As complexidades foram derivadas dos pseudo algoritmos previamente apresentados. Todas elas envolveram análises muito simples (loops dentro de loops), de forma que apenas os resultados finais serão apresentados.

Algorithm 2: ZicaZeroAnelDual - Programação dinâmica

Input: \mathbb{V} , matrizCobertura
Output: \mathbb{V}'

```
1  $\mathbb{V}' = [];$ 
2 for  $i \leftarrow 1$  to  $\mathbb{V}.tamanho$  do
3   for  $j \leftarrow 1$  to  $\mathbb{V}.tamanho$  do
4     if  $i == j$  then
5        $matriz[i][j][focosCobertos[\mathbb{V}[j]][0]] = True;$ 
6        $matriz[i][j][focosCobertos[\mathbb{V}[j]][1]] = True;$ 
7        $colunaEsq = j;$ 
8     else
9        $matriz[i][j] = matriz[i][colunaEsq];$ 
10       $matriz[i][j][focosCobertos[\mathbb{V}[j]][0]] = True;$ 
11       $matriz[i][j][focosCobertos[\mathbb{V}[j]][1]] = True;$ 
12 for  $i \leftarrow 1$  to  $\mathbb{V}.tamanho$  do
13    $linhaAtual = 0;$ 
14    $colunaAtual = i;$ 
15   for  $j \leftarrow 1$  to  $\mathbb{V}.tamanho$  do
16     if todas as posições do vetor de cobertura estão marcadas como True then
17        $posInicial = linhaAtual;$ 
18        $posFinal = colunaAtual;$ 
19        $numVoluntarios = calculaNumVoluntarios(posInicial, posFinal);$ 
20       for  $k \leftarrow 1$  to  $numVoluntarios$  do
21          $\mathbb{V}' = \mathbb{V}' + \mathbb{V}[(posInicial + k) \bmod \mathbb{V}.tamanho]$ 
```

4.1. Busca por Força-bruta

4.1.1. Complexidade de tempo

$$O(n \cdot n \cdot (\frac{n}{2} \cdot k)) = O(n^3 \cdot k)$$

4.1.2. Complexidade de espaço

$$O(n + 2n + n + k) = O(n + k)$$

4.2. Programação Dinâmica

4.2.1. Complexidade de tempo

$$O(n \cdot n + n \cdot n \cdot k)) = O(n^2 \cdot k)$$

4.2.2. Complexidade de espaço

$$O(n + 2n + n^2 \cdot k) = O(n^2 \cdot k)$$

4.3. Algoritmo Guloso

4.3.1. Complexidade de tempo

$$O(n \cdot (n - 1) \cdot 2k)) = O(n^2 \cdot k)$$

4.3.2. Complexidade de espaço

$$O(n + 2n + k + n^2 + n) = O(n^2 \cdot k)$$

5. Implementação (exercício 3)

O trabalho foi implementado em Python, em ambiente Linux. 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 4 e 5)

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.

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 não exponencial no tempo de execução (Figura 1).

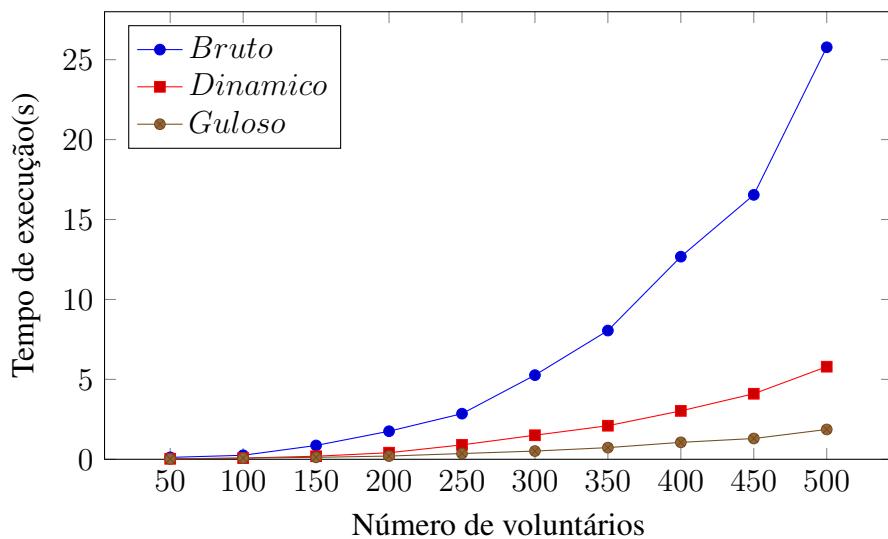


Figura 1. Tempo de execução em função do número de voluntários

- Impacto do número de focos no tempo de execução: a Figura 2 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 os algoritmos.
- Impacto do número de voluntários na quantidade de memória alocada: a Figura 3 mostra que o número de voluntários tem impacto quadrático nas abordagens dinâmica e gulosa.
- Impacto do número de focos na quantidade de memória alocada: a Figura 4 mostra que o número de voluntários tem impacto linear em todas as abordagens.

7. Conclusão

O programa conseguiu realizar corretamente a tarefa proposta, independentemente da combinação de valores para os fatores da entrada. O tempo de execução e a quantidade de memória necessária para a execução do algoritmo ficaram dentro do esperado. Algumas saídas diferiram do conjunto de saídas fornecidas, uma vez que o critério de desempate é diferente. Ainda assim o algoritmo sempre chegou ao final da execução com o resultado esperado, o que era o objetivo principal do trabalho.

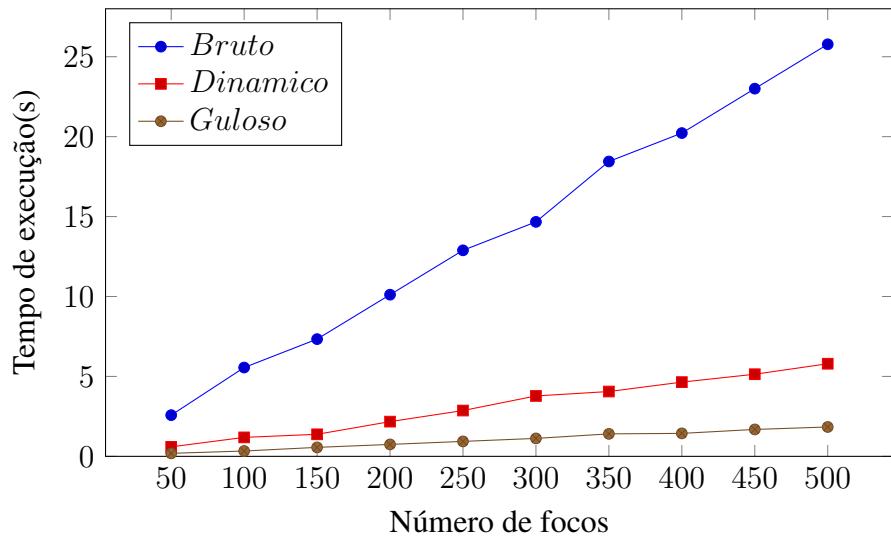


Figura 2. Tempo de execução em função do número de focos

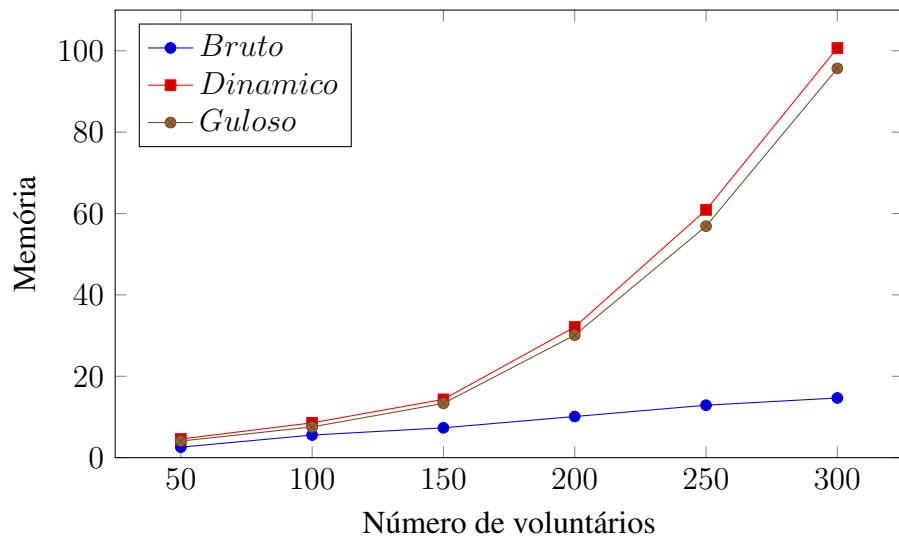


Figura 3. Memória utilizada em função do número de voluntários

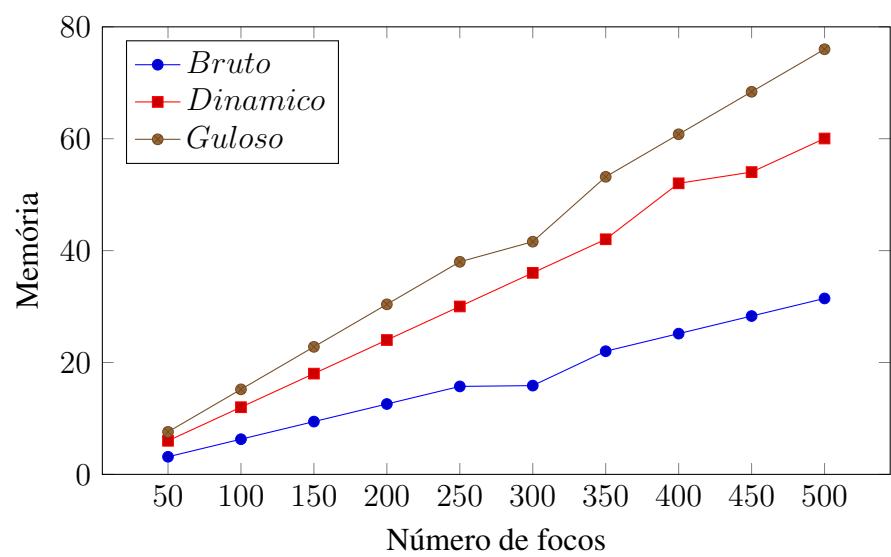


Figura 4. Memória utilizada em função do número de focos

Referências

- [1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. “*Introduction to Algorithms*”. McGraw-Hill Higher Education, 2001.

Algorithm 3: ZicaZeroAnelDual - Abordagem gulosa

Input: \mathbb{V} , matrizCobertura

Output: \mathbb{V}'

```
1  focosCobertos = [False, False, ..., False];
2  conjuntosVoluntarios = [[], [], ..., []];
3  numVoluntariosCjto = [0, 0, ..., 0];
4   $\mathbb{V}' = []$ ;
5  for  $i \leftarrow 1$  to  $\mathbb{V}.\text{tamanho}$  do
6    voluntarioInicial =  $\mathbb{V}[i]$ ;
7    conjuntosVoluntarios[ $i$ ] = conjuntosVoluntarios[ $i$ ] + voluntarioInicial;
8    numVoluntariosCjto[ $i$ ]++;
9    focosCobertos[matrizCobertura[ $\mathbb{V}[\text{voluntarioInicial}]$ ][0]] = True;
10   focosCobertos[matrizCobertura[ $\mathbb{V}[\text{voluntarioInicial}]$ ][1]] = True;
11   voluntarioEsq =  $\mathbb{V}[(i - 1) \bmod \mathbb{V}.\text{tamanho}]$ ;
12   voluntarioDir =  $\mathbb{V}[(i + 1) \bmod \mathbb{V}.\text{tamanho}]$ ;
13   for  $j \leftarrow 1$  to  $\mathbb{V}.\text{tamanho} - 1$  do
14     if todas as posições do vetor de cobertura estão marcadas como True then
15       break;
16     numNovosFocosEsq =
17       numNovosFocos(focosCobertos, matrizCobertura, voluntarioEsq);
18     numNovosFocosDir =
19       numNovosFocos(focosCobertos, matrizCobertura, voluntarioDir);
20     if numNovosFocosEsq > numNovosFocosDir then
21       proxVoluntario = voluntarioEsq;
22       voluntarioEsq =  $\mathbb{V}[(\mathbb{V}.\text{indice}(\text{proxVoluntario}) - 1) \bmod \mathbb{V}.\text{tamanho}]$ ;
23     else
24       proxVoluntario = voluntarioDir;
25       voluntarioDir =  $\mathbb{V}[(\mathbb{V}.\text{indice}(\text{proxVoluntario}) + 1) \bmod \mathbb{V}.\text{tamanho}]$ ;
26   conjuntosVoluntarios[ $i$ ] = conjuntosVoluntarios[ $i$ ] + proxVoluntario;
27   numVoluntariosCjto[ $i$ ]++;
28
29  $\mathbb{V}' =$ 
30   conjuntosVoluntarios[numVoluntariosCjto.indice(min(numVoluntariosCjto))];
31 return  $\mathbb{V}'$ ;
```

Trabalho Pratico: ZikaZeroAnelDual

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. Proponha algoritmos de Complexidade Assintótica Polinomial com o Tamanho da entrada (n , m e r) para resolver o Problema ZikaZeroAnelDual usando os seguintes paradigmas:

1. Busca por Força-bruta

Dado como entrada um grafo $G(V, A)$, em que V é o conjunto de n voluntários e A é o conjunto de $m = |V|$ laços de amizade e um conjunto F dos r focos de reprodução do mosquito, primeiramente é gerado para o algoritmo por força bruta todas as combinações de vértices conexos existente para o conjunto de intervalo $\{s \in Z | 1 \leq s \leq |V|\}$, sem que ocorra casos de repetição, sendo definido um vetor para representar os laços de amizade de forma sequencial sempre a partir do primeiro vértice v_1 . Posteriormente o vetor é percorrido com janelas de tamanho crescente, onde para cada iteração é verificado se o conjunto de vértices dentro da janela de amostragem atual consegue cobrir todos os focos. Por ultimo, o menor conjunto de vértices conexos pertencentes ao vetor de combinações que conseguem atingir todos os focos são adicionados a um novo vetor de *minimum subsets*, em seguida este vetor contendo o resultado final é ordenado pelo algoritmo *bubble sort* e o resultado então sera salvo no arquivo de saída, finalizando assim o programa.

2. Programação Dinâmica

Dado como entrada um grafo $G(V, A)$, em que V é o conjunto de n voluntários e A é o conjunto de $m = |V|$ laços de amizade e um conjunto F dos r focos de reprodução do mosquito, primeiramente é gerada para o algoritmo uma tabela $T[i][X]$ equivalente ao número mínimo de conjuntos de $\{S_1, S_2, \dots, S_i\}$ que cobre X para cada $X \subseteq U$ e $i \in \{1, 2, \dots, |V|\}$ [Cover], sendo X composto por todas as combinações de vértices conexos existente para o conjunto de intervalo $\{s \in Z | 1 \leq s \leq |V|\}$. Posteriormente a tabela é preenchida respeitando as seguintes restrições:

$$\begin{aligned} T[0][\emptyset] &= 0 \\ T[0][X \neq \emptyset] &= \infty \\ T[i][X] &= \min \left\{ \begin{array}{l} 1 + T[i-1][X \setminus S_i] \\ T[i-1][X] \end{array} \right. \end{aligned}$$

A tabela $T[i][X]$ irá armazenar em suas células a quantidade mínima de vértices, considerando a inclusão de i-ésimos vértices por linha ta tabela, para cobrir os conjuntos formados pela coluna X . O conjunto solução final poderá ser

encontrado na posição $T[m][U]$, no entanto, a tabela T armazena apenas a menor quantidade de vértices necessário para satisfazer as restrições impostas, uma segunda tabela terá que ser mantida para mapear exatamente quais vértices foram utilizados para satisfazer as condições de cada célula da tabela principal. Por ultimo, o menor conjunto de vértices conexos pertencentes a tabela é resgatado e são adicionados a um novo vetor de *minimum subsets*, em seguida este vetor é ordenado em ordem crescente pelo algoritmo *bubble sort* e o resultado então sera salvo no arquivo de saída, finalizando assim o programa.

3. Algoritmo Guloso

Dado como entrada um grafo $G(V, A)$, em que V é o conjunto de n voluntários e A é o conjunto de $m = |V|$ laços de amizade e um conjunto F dos r focos de reprodução do mosquito, primeiramente é gerado um vetor para representar os laços de amizade de forma sequencial sempre a partir do primeiro vértice v_1 . Posteriormente dois índices são criados com o intuito de percorrer o este vetor em direções opostas de conectividade, horário e anti-horário. Dois vetores temporários também são criados para armazenar o subconjunto de vértices para da uma das duas possibilidades de inclusão do vértice adjacente como melhor solução local. Em seguida, é calculado para cada um dos dois subconjunto a quantidade de focos respectivamente cobertos por cada alternativa, o subconjunto de maior cobertura é então selecionado e o índice respectivo a este é utilizado para incluir o vértice no vetor solução de *minimum subsets*. Para cada iteração é verificado se os vértices pertencente ao vetor solução conseguem cobrir o conjunto de foco $|F|$, em caso positivo, o vetor de *minimum subsets* é ordenado pelo algoritmo *bubble sort*. Por ultimo, os vértices pertencente ao conjunto resultado ordenados de forma crescente são salvos em um arquivo de saída, finalizando assim o programa.

Exercício 2. Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) dos algoritmos propostos no Exercício 1.

1. Busca por Força-bruta

Complexidades Temporal: Nesse caso, o vetor contendo todas as combinações conexas existente no grafo deve ser percorrido por uma janela crescente a cada iteração, até que todos os focos sejam cobertos. Para essa condição temos a janela de busca sempre percorre um espaço relativo a quantidade total de vértices $|V|$, e no pior caso a quantidade de iterações também será igual ao tamanho máximo de vértices $|V|$, sendo a complexidade resultante $|V| * |V|$ igual a $O(|V|^2)$.

Complexidades Espacial: Nesse caso, o tamanho do vetor de combinações domina a complexidade espacial do algoritmo, sendo $O(n)$.

2. Programação Dinâmica

Complexidades Temporal: Nesse caso, a tabela $T[i][X]$ contendo todas as melhores soluções parciais deve ser preenchida com base nas restrições impostas referente as linha i e colunas X . Para essa condição, em que a tabela não é quadrática, temos uma complexidade $O(|i| * |X|)$.

Complexidades Espacial: Nesse caso, o tamanho da tabela domina a complexidade espacial do algoritmo, sendo necessário a utilização de uma tabela auxiliar

e mesmo tamanho para recuperação do resultado final, temos uma complexidade de duas vezes o tamanho total da tabela $O(|T|^2)$.

3. Algoritmo Guloso

Complexidades Temporal: Nesse caso, o vetor contendo todas as relações sequenciais conexas dos vértices existente no grafo deve ser percorrido por dois índices de extremidade oposta, a cada iteração um dos dois índices é adicionado como pertencente à solução final. Para essa condição temos uma complexidade $O(|V|)$, entretanto, para o pior caso em que todos os conjunto de saída poderão estar desordenados, a complexidade temporal é dominada pelo algoritmo *bubble sorte* $O(n^2)$.

Complexidades Espacial: Nesse caso, o tamanho do vetor de vértices conexo domina a complexidade espacial do algoritmo, sendo também $O(|V|)$.

Exercício 4. Execute testes da implementação do Exercício 3, propondo instâncias interessantes para o Problema ZikaZeroAnelDual. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n, m e r) comparando os três algoritmos implementados.

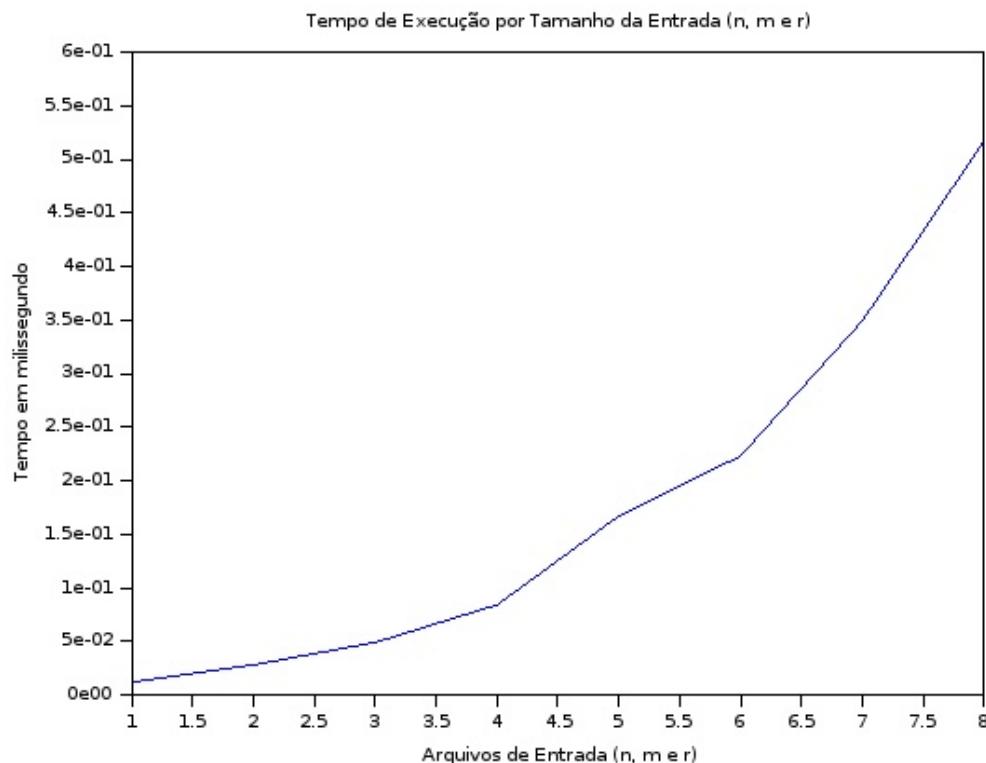


Figura 1. Análise da complexidade temporal em milissegundo do algoritmo de força bruta para os conjuntos de entrada (n,m,r) $\{\{3,3,4\}, \{4,4,5\}, \{5,5,6\}, \{6,6,7\}, \{7,7,8\}, \{8,8,9\}, \{9,9,10\}, \{10,10,11\}\}$

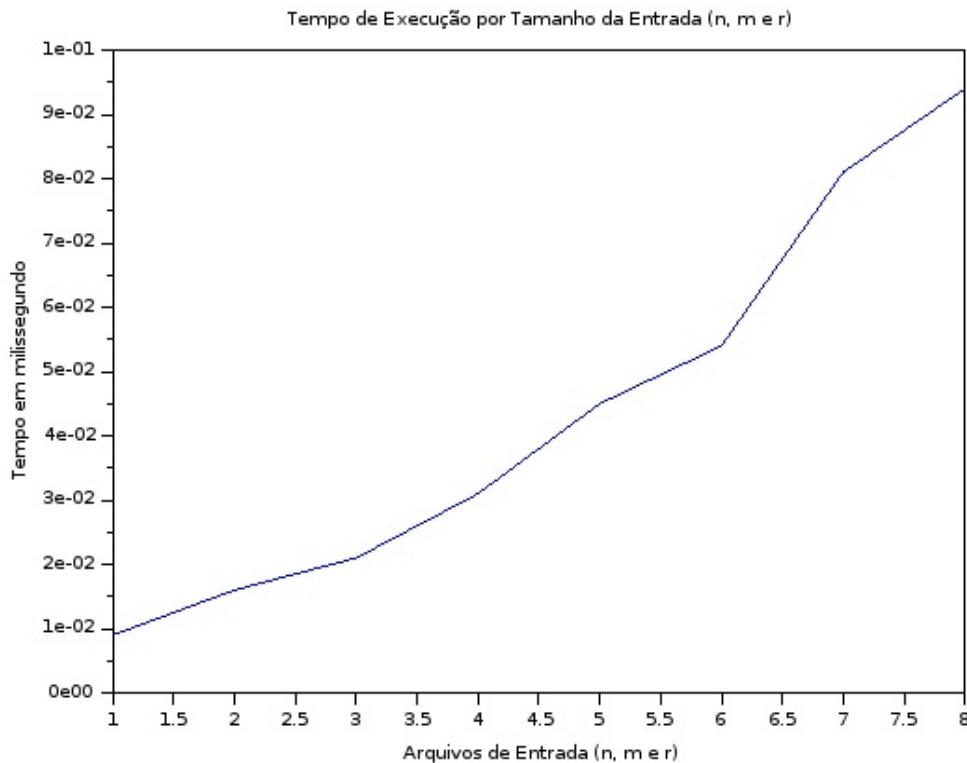


Figura 2. Análise da complexidade temporal em milissegundo do algoritmo guloso para os conjuntos de entrada (n,m,r) $\{\{3,3,4\}, \{4,4,5\}, \{5,5,6\}, \{6,6,7\}, \{7,7,8\}, \{8,8,9\}, \{9,9,10\}, \{10,10,11\}\}$

Exercício 5. Compare a análise e a execução, respectivamente, Exercícios 2 e 4. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

Como podemos visualizar nos gráficos apresentados no exercício 4 para os algoritmos de força bruta e guloso, a complexidade temporal observada em ambas as abordagens em função do arquivo de entrada (m, n, r), estão em concordância com a previsão realizada no exercício 2 por intermédio da análise assintótica. Como podemos observar melhor pela tabela a seguir, o algoritmo de força bruta possui o pior desempenho temporal, apresentando um comportamento de acréscimo muito superior ao algoritmo guloso, que manteve o seu crescimento temporal praticamente constante.

	1	2	3	4	5	6	7	8
Força Bruta	0.012	0.028	0.049	0.084	0.167	0.223	0.349	0.517
Guloso	0.009	0.016	0.021	0.031	0.045	0.054	0.081	0.094

1. References

Referências

Cover, S. Dynamic programming, inclusion-exclusion principle and fsc (fast subseteq convolution).

Projeto e Análise de Algoritmos - Grafos AnelDual

Arthur Barbosa Câmara

Maio 2016

1 Introdução

Esse trabalho tem como objetivo a modelagem e análise de algoritmos em grafos, usando diferentes paradigmas de programação. Especificamente, foi modelado e resolvido o seguinte problema: Dado um grafo Anel, onde os voluntários são os vértices, onde todo vértice se conecta a dois outros, formando uma anel, e o conjunto de focos do mosquito *Aedes Aegypti* que cada voluntário pode alcançar, selecionar o menor subconjunto de voluntários que alcancem todos os focos e que, ao mesmo tempo, forte uma rede de colaboração coesa. Ou seja, o subgrafo induzido pelos voluntários é conexo.

Nas próximas seções, a modelagem do problema, assim como os detalhes de diferentes implementações serão mostrados. Ao final, uma análise dos experimentos realizados também será discutida.

2 Modelagem

O problema pode ser modelado como um grafo não direcionado e não ponderado $G = (V, E)$ da seguinte maneira:

$V \rightarrow$ O conjunto de voluntários é representado pelos vértices de um grafo

$E \rightarrow$ As relações de amizade existentes entre os voluntários

Para a implementação, foi escolhida uma representação em Lista de Adjacências, o que facilita a checagem se um subgrafo é ou não conexo e a iteração sobre o conjunto de vértices vizinhos

Além disso, as relações entre usuários e focos são modeladas como um vetor de *unordered_set* em C++, o que possibilita uma checagem rápida (em $O(1)$) se o voluntário i pode ou não acessar o foco f

Na solução do problema, assume-se ainda que cada voluntário tem acesso a pelo menos um foco. Isso garante que o problema tenha no mínimo uma solução (quando cada voluntário tem apenas um foco distinto)

3 Algoritmos

Para esse problema, foram desenvolvidas diferentes abordagens para a solução. Uma força bruta e uma gulosa. Foi assumido que, como o grafo é um anel, ele sempre será conectado se os vértices forem adjacentes entre si.

Por tanto, a checagem de conectividade do grafo é explícita, e exploramos esse fato para facilitar a implementação.

3.1 Cobertura de focos

Para cada possível solução do problema, é necessário verificar se ela de fato resolve o problema, ou seja, se essa combinação de voluntários cobre ou não todos os focos. O Algoritmo 1 mostra como isso é feito. O algoritmo mantém um conjunto com os focos já visitados, e, para cada voluntário da combinação, todos os focos que ele pode alcançar são adicionados ao conjunto. A execução termina se todos os focos já foram visitados ou se todos os voluntários já foram testados.

Algorithm 1: Cobertura de Focos

```
input : Conjunto  $v$  de voluntários da combinação
output: True ou False, se a combinação cobre todos os vértices
cobertos = {};
foreach  $i$  em  $v$  do
    foreach  $f$  na lista de focos de  $i$  do
        | cobertos = cobertos  $\cup$   $f$ 
    end
end
if tamanho de cobertos == número de focos then
    | retorne True;
else
    | retorne False;
end
```

3.2 Algoritmo de Força bruta

Para o algoritmo de força bruta, foi utilizada uma implementação ingênuia, onde, a partir de cada vértice i , todas as possíveis soluções, iniciando de i até o vértice à direita de i , são testadas, incrementando a solução em um elemento até completar a volta no anel.

Vale notar ainda que, quando, a partir do vértice i , se encontra alguma solução para o problema, essa é, necessariamente, a melhor possível se iniciando desse vértice, e a iteração para.

Algorithm 2: Algoritmo Força Bruta

input : Grafo Anel G e Conjunto F de focos
output: Vetor com os vértices da solução

```
S := [ ];
i := -1;
melhor := [ ];
while  $i \neq 1$  do
    if  $i = -1$  then
        |   i := 1;
    end
    j := i;
    while  $|S| \leq num\_vertices$  do
        |   S := S + j;
        |   if ( $F.FocusCover(S) = |F|$ ) e ( $|S| \leq |melhor|$ ) then
        |       |   ;                                     // Algoritmo 1
        |       |   melhor := S;
        |       |   break;
        |   end
        |   j := j  $\rightarrow$  direita;
    end
    i := i  $\rightarrow$  direita
end
retorne melhor;
```

3.3 Guloso

O algoritmo guloso também tem uma abordagem simples, mas eficiente. A cada passo, ele cresce a solução para a esquerda ou direita, escolhendo qual dos lados o conjunto de focos cobertos cresce mais. A abordagem está no algoritmo 3. Note que, por ser uma heurística, a abordagem gulosa não necessariamente retorna resultado ótimo.

Algorithm 3: Algoritmo Guloso

```
input : Grafo Anel  $G$  e Conjunto  $F$  de focos
output: lista com os vértices da solução

 $S := []$ ;
 $i :=$  Voluntário com maior número de focos;
 $FC := []$ ; // Focos já cobertos
while  $|FC| < |F|$  do
     $esquerda := |FC| + F[S.front \rightarrow esquerda]$ ;
     $direita := |FC| + F[S.back \rightarrow direita]$ ;
    if  $|esquerda| > |direita|$  then
         $FC := esquerda$ ;
         $S := S.front -> esquerda + S$ ;
    else
         $FC := direita$ ;
         $S := S + S.back -> direita$ ;
    end
end
retorne  $S$ ;
```

4 Complexidade

Nessa sessão, a complexidade de tempo e espacial do problema e das abordagens será analisada.

Nessa sessão, usaremos n como o número de voluntários e k como o número de focos no problema.

4.1 Complexidade temporal

4.1.1 Cobertura de Focos

Dada uma possível solução, o próximo passo é checar se ela consegue cobrir todos os focos. Pelo **Algoritmo 1**, podemos ver que isso é feito em dois loops. O primeiro itera sobre todos os n voluntários da combinação. O segundo loop itera sobre cada um dos k focos que o voluntário tem acesso. O conjunto de vértices checados é implementado como um *unordered_set* e, portanto, possui operações em tempo constante para busca e inserção.

Por conta dessas duas iterações aninhadas, a checagem de cobertura de focos é executada em $O(n \times k)$ operações

4.2 Força Bruta

Graças a restrição do problema a um grafo anel, a solução do problema deixa de ser exponencial e passa a ser polinomial.

Pelo algoritmo 2 , podemos levantar o seguinte:

- O algoritmo executa 2 loops aninhados
- cada loop pode, no pior caso, passar por todos os vértices até completar a volta no anel
- Dentro do loop inferior, é verificada se a solução que inicia em i e termina em j cobre todos os focos

Pelo algoritmo 1, sabemos que o último item possui complexidade $O(n \times k)$. Como isso é executado $n \times n = n^2$ vezes, podemos concluir que o algoritmo de força bruta possui complexidade $O(k \times n^3)$.

4.3 Guloso

Observando o algoritmo 3, podemos identificar que:

- Cada iteração, o algoritmo gera duas possíveis soluções
- A união de dois conjuntos tem complexidade $O(k)$, e é executada duas vezes.
- Apenas uma iteração é realizada, escolhendo apenas uma solução a cada passo, a que maximiza localmente a solução.

Como apenas uma iteração é realizada, e cada uma possui complexidade $O(k)$, podemos deduzir que a complexidade do algoritmo é $O(nk)$, o que é consideravelmente menor do que o algoritmo de força bruta.

4.4 Complexidade espacial

As estrutura de dados básicas do problema é o grafo de amizades e o conjunto de voluntários com a lista de focos que cada um pode atingir. A complexidade do primeiro é de $O(n)$, já que o número de arestas é constante ($\frac{2n}{2} = n$). Do segundo, como são n voluntários, com k focos, temos a complexidade de $O(m \times k)$

Cada parte do problema cria uma outra estrutura de dados adicional, mas que é reutilizada, ou seja, o gasto de memória não aumenta a cada iteração.

Para o algoritmo de cobertura de focos, é criada uma estrutura adicional, com tamanho $O(k)$.

Para o Força bruta, apenas a resposta, com tamanho $O(n)$ é criado.

Para o guloso, além da solução com tamanho $O(n)$, a cada iteração são criados mais dois vetores com as possíveis soluções de cada lado, com complexidade $O(k)$ em cada um. Combinando essas duas complexidades, temos ao fim uma complexidade espacial de $O(n+k)$, já que os vetores candidatos são reutilizados.

5 Experimentos

Como visto anteriormente, diferentemente do problema anterior, nesse caso a variável com maior peso nas análises é o número de vértices, e não somente o número de vértices, apesar de ambos os fatores possuírem impacto nas respostas.

Os experimentos foram executados com o seguinte comando de compilação:

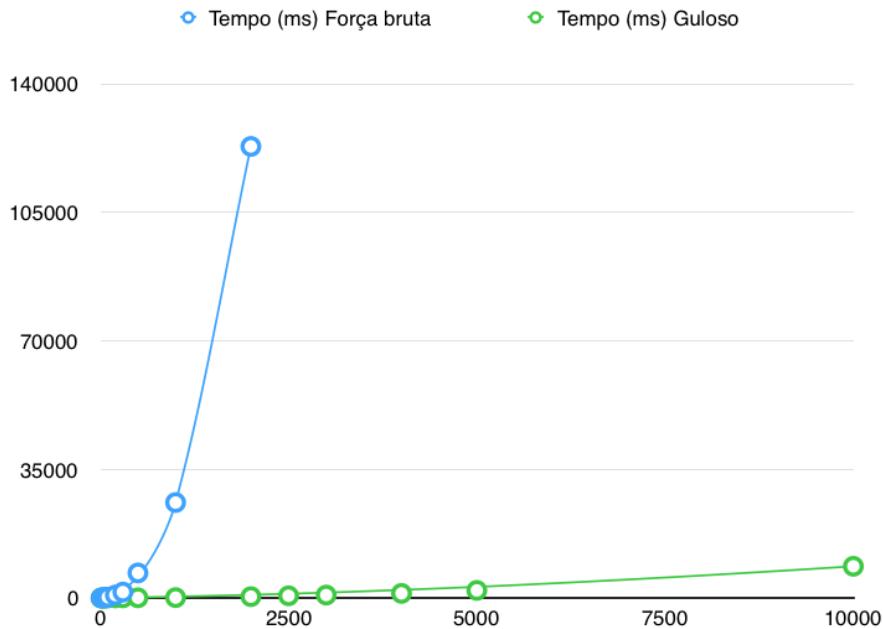
```
g++ -Wall -g -std=c++14 -O3 <arquivos cpp e hpp>
```

Para cada variação, foi medida pelo menos 10 execuções do código em diferentes grafos (com as mesmas características) e a média do tempo de execução foi calculada.

A máquina usada para teses possui SO Mac OS X 10.11.3, com processador Intel Core i5 de 1.6GHz e 4GB de memória DDR3 de 1600MHz. O compilador usado foi o Apple LLVM 7.3.0.

O gráfico de tempo de execução mostra claramente que, mesmo sendo polinomial, o algoritmo força bruta é consideravelmente pior do que o guloso, para grafos com a mesma configuração. É claro como a potência de 3 interfere significativamente no resultado. Note que, para o força bruta, foram omitidos resultados com grafos muito grandes. Isso foi feito para que a curva do guloso fosse visível.

Figure 1: Variação do tamanho do grafo



Com o gráfico, fica claro que, efetivamente, a solução força bruta, apesar de possuir complexidade polinomial, e conseguir sempre um resultado ótimo, possui uma complexidade muito superior ao do guloso na prática, por ser linear.

Portanto, recomenda-se, para grafos com muitos vértices, que a solução gulosa seja utilizada, já que ainda consegue um resultado consideravelmente bom.

Relatório do Trabalho Prático de Paradigmas - PAA

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

2016/1

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 1 - 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.

A fim de explorar o grafo em anel (cada vértice possui grau 2 e o grafo é conexo), após ler o grafo, antes de executar os algoritmos para encontrar a rede de colaboração, um vetor é construído contendo os vértices vizinhos, em sequência, de modo que v_i, v_{i+1} sejam vizinhos no grafo original, para qualquer $i \in 1..|\mathcal{G}|$. A construção desse vetor tem custo de espaço $\mathcal{O}(|\mathbb{V}|)$ e de execução $\mathcal{O}(|\mathbb{V}| + |\mathbb{E}|) = \mathcal{O}(|\mathbb{V}|)$ (por ser um grafo anel). O resultado final é obtido consultando os índices originais dos vértices do grafo \mathcal{G} a partir da sequência **contígua** $v_i..v_j$.

Brute-Force-Solve

A idéia da solução por força bruta é (1) gerar todos os candidatos a solução, cada candidato sendo gerado exatamente uma vez e (2) verificar se cada candidato é uma solução. Para isso, cada conjunto contíguo de 1, 2, 3, ..., $|\mathcal{G}|$ elementos é avaliado a fim de encontrar um subconjunto que cubra todos os $|\mathbb{F}|$ focos.

```
BRUTE-FORCE-SOLVE( $\mathcal{G}, \mathbb{F}$ )
1: for solution_size = 1.. $|\mathcal{G}|$  do
2:   for  $i = 1..|\mathcal{G}|$  do
3:      $\mathbb{S} = \emptyset$ 
4:     for  $j = i..(i+solution\_size)$  do
5:        $\mathbb{S} \leftarrow \mathbb{S} \cup \mathcal{F}(v_i)$ 
6:     end for
7:     if  $\mathbb{S} = \mathbb{F}$  then
8:       return  $(v_i, v_j)$ 
9:     end if
10:   end for
```

11: **end for**

Greedy-Solve

A idéia da solução gulosa é explorar conjuntos que maximizem o número de focos cobertos. A união máxima entre dois subconjuntos de focos é computada de maneira gulosa a cada iteração. Ela é definida pela união dos subconjuntos de focos do conjunto de vértices atual com o conjunto de vértices anterior. Esse processo é repetido a cada iteração (totalizando $|\mathcal{G}|$ iterações). Ao encontrar a união que resulte em todos os focos cobertos, a solução foi encontrada.

```
GREEDY-SOLVE( $\mathcal{G}, \mathbb{F}$ )
1:  $\mathbb{S} \leftarrow |\mathcal{G}| \times |\mathcal{G}|$  matrix filled with  $\emptyset$ 
2: for solution_size = 1.. $|\mathcal{G}|$  do
3:   max = 0
4:    $v_i^{(max)} = nil$ 
5:   for i =  $|\mathcal{G}|..1$  do
6:      $\mathbb{S}[i][solution\_size] = \mathbb{S}[i][solution\_size-1] \cup \mathbb{S}[i+1][solution\_size-1]$ 
7:     if max <  $|\mathbb{S}[i][solution\_size]|$  then
8:       max =  $|\mathbb{S}[i][solution\_size]|$ 
9:        $v_i^{(max)} = i$ 
10:    end if
11:   end for
12:   if max =  $|\mathbb{F}|$  then
13:     return ( $v_i^{(max)}, v_{i+solution\_size}^{(max)}$ )
14:   end if
15: end for
```

Dynamic-Programming-Solve

A idéia da solução por programação dinâmica é determinar o número mínimo de vértices necessários para cobrir um determinado subconjunto de focos. A explicação desse algoritmo se dará em duas partes:

Vértices

O que queremos encontrar é, para o conjunto contíguo de vértices de v_i a v_j , o número mínimo de vértices que cobre um conjunto contíguo de focos f_i a f_j . Inicialmente, esse número é infinito, e permanece infinito enquanto não seja possível cobrir todos os focos de f_i a f_j utilizando somente os vértices de v_i a v_j .

Trivialmente, se $v_i = v_j$, o número mínimo de vértices que cobre $f_i = f_j \in \mathcal{F}(v_i)$ é 1.

Focos

O número de vértices para cobrir qualquer subconjunto contíguo de focos $f_i..f_j$ para um subconjunto contíguo de vértices $v_i..v_j$ não pode ser superior aos custos de $v_{i+1}..v_j$ e $v_i..v_{j-1}$ individualmente. Analogamente, não pode ser menor, a não ser que não fosse coberto por nenhum dos subconjuntos de vértices $v_{i+1}..v_j$ e $v_i..v_{j-1}$. Neste último caso, o número mínimo de vértices necessários para cobrir o conjunto de focos vai ser exatamente igual ao número de vértices de $v_i..v_j$, ou seja, igual a $j - i + 1$.

*Note que a ordem de i e j no algoritmo está invertida com relação à da presente descrição. O motivo disso é a utilização da diagonal inferior da matriz para otimizar a localidade de referência. Os resultados, entretanto, são equivalentes. Note também que, para evitar resultados incorretos quando a solução real passa pelo vértice escolhido para ser v_0 , dobramos o número de vértices. Assim, a solução 6, 0, 1, por exemplo, ainda pode ser representada como uma sequência monotonicamente crescente: 6, 7, 8).

DYNAMIC-PROGRAMMING-SOLVE(\mathcal{G}, \mathbb{F})

```

1:  $\mathbb{S} \leftarrow 2|\mathcal{G}| \times 2|\mathcal{G}| \times |\mathbb{F}| \times |\mathbb{F}|$  matrix filled with  $\infty$ 
2: for  $v_i = 1..2|\mathcal{G}|$  do
3:   for  $v_j = i$  downto 1 do
4:     if  $v_i = v_j$  then
5:       for  $f \in \mathcal{F}(v_{v_i})$  do
6:          $\mathbb{S}[v_i][v_j][f][f] = 1$ 
7:       end for
8:     end if
9:     for  $f_i = 1..|\mathbb{F}|$  do
10:    for  $f_j = f_i$  downto 1 do
11:      if  $v_i \neq v_j$  then
12:         $\mathbb{S}[v_i][v_j][f_i][f_j] = \min(\mathbb{S}[v_i - 1][v_j][f_i][f_j], \mathbb{S}[v_i][v_j + 1][f_i][f_j])$ 
13:      end if
14:      if  $f_i \neq f_j$  then
15:         $\mathbb{S}[v_i][v_j][f_i][f_j] = \min(\mathbb{S}[v_i][v_j][f_i][f_j],$ 
            $\max(v_i - v_j + 1,$ 
            $\max(\mathbb{S}[v_i][v_j][f_i - 1][f_j], \mathbb{S}[v_i][v_j][f_i][f_j + 1])))$ 
16:      end if
17:    end for
18:  end for
19: end for
20: end for

```

Exercício 2 - Análise de Complexidade

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

Brute-Force-Solver

O algoritmo de solução por força bruta, bem como todos os outros, mantém um vetor com os vértices ordenados pela sequência de vizinhos. Este passo de inicialização possui custo assintótico de tempo e espaço $\mathcal{O}(|\mathbb{V}|)$.

Após esse passo, para cada tamanho válido de solução (de 1 a $|\mathbb{V}|$), para cada vértice inicial (de 1 a $|\mathbb{V}|$) e final (de 1 a $|\mathbb{V}|$) é calculada a união dos subconjuntos de focos cobertos pela união entre os vértices inicial e final (incluindo todos os vértices intermediários). A complexidade assintótica de tempo de execução é, consequentemente, $\mathcal{O}(|\mathbb{F}||\mathbb{V}|^3)$. Para cada subconjunto de vértices considerado, o conjunto resultante de focos é mantido para confirmar/refutar a validade da solução atual. Somado à complexidade de espaço do passo de inicialização, a complexidade de espaço resulta em $\mathcal{O}(|\mathbb{V}| + |\mathbb{F}|)$.

Greedy-Solver

Para cada tamanho válido de solução (de 1 a $|\mathbb{V}|$), para cada subconjunto de vértices inicial (de 1 a $|\mathbb{V}|$), é feita a união entre dois subconjuntos de focos (do subconjunto inicial e do seguinte). A complexidade assintótica de tempo de execução é, consequentemente, $\mathcal{O}(|\mathbb{F}||\mathbb{V}|^2)$, considerando que os subconjuntos de focos são mantidos em uma tabela hash. O custo de espaço é $\mathcal{O}(|\mathbb{F}||\mathbb{V}|^2)$, que pode ser reduzido para $\mathcal{O}(|\mathbb{F}||\mathbb{V}|)$ aproveitando o fato de que a única geração necessária em memória para calcular $\mathbb{S}[i][\text{solution_size}]$ é a correspondente a $\text{solution_size}-1$, as demais podendo ser descartadas.

Dynamic-Programming-Solver

Do algoritmo, decorre trivialmente que a complexidade assintótica de execução e espaço é $\mathcal{O}(|\mathbb{V}|^2|\mathbb{F}|^2)$.

Exercício 4 - 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 respeitando o número de vértices n (e consequentemente de arestas $m = n$), e focos r .
- Ser capaz de gerar instâncias cujo tamanho da solução seja, aproximadamente, um parâmetro k ;

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

1. Gerar os n vértices.
2. Criar um grafo anel com os n vértices escolhidos.
3. Escolher uma posição (v_i) para ser o início da solução aproximada de k vértices.
4. No primeiro vértice da solução aproximada, colocar o primeiro foco (F_1) e mais um foco escolhido aleatoriamente como focos aos quais o vértice tem acesso.
5. No último vértice da solução aproximada ($v_j = v_{i+k}$), colocar último foco F_r e mais um foco escolhido aleatoriamente como focos aos quais ele tem acesso.
6. Nos vértices entre o primeiro da solução aproximada e os k seguintes, colocar em round-robin os focos ($F_i : i \in \{2..r-1\}$), como focos aos quais eles têm acesso.
7. 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.

A prova de que o gerador funciona segue dos fatos de que: Desde que k seja no mínimo a metade de f , todos os focos serão cobertos pela solução aproximada, de modo que garantidamente existirá uma solução de no máximo tamanho k . Quando $n > 2k$, então a melhor solução tem garantidamente também tamanho k (v_i e v_j necessariamente estão na solução). Quando $n \leq 2k$, pode ser que exista uma solução em que v_i e v_j estão incluídos, mas o tamanho da solução é menor do que k . Por isso o gerador é aproximado se não forem impostas restrições quanto ao tamanho de n e k , e exato se for imposto $n > 2k$. ■

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

Exemplos de Instâncias Geradas

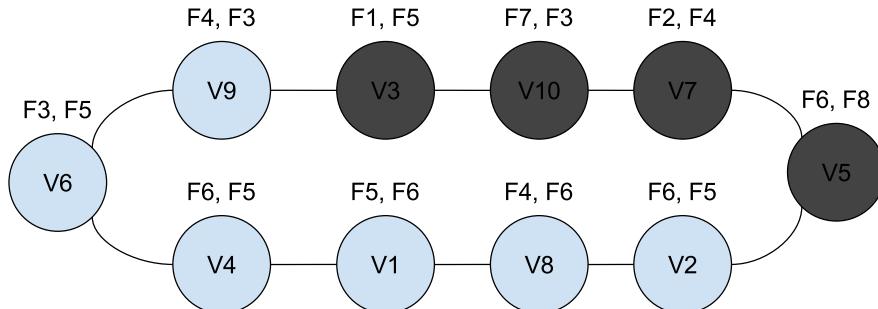


Figure 1: Exemplo de instância gerada para $n = 10, r = 8, k = 4$

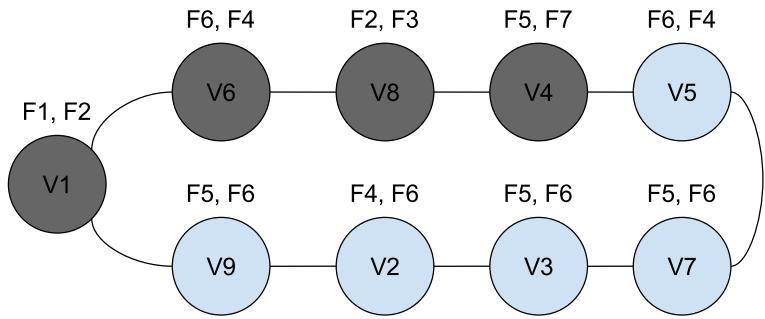


Figure 2: Exemplo de instância gerada para $n = 9, r = 7, k = 4$

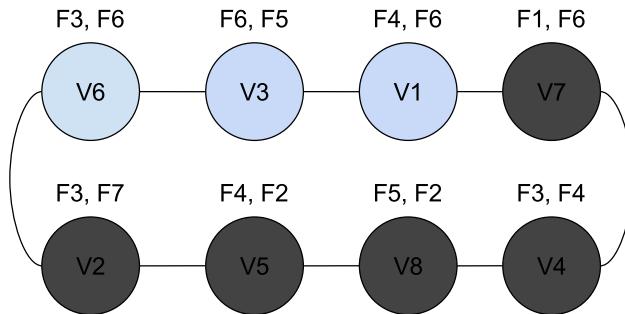


Figure 3: Exemplo de instância gerada para $n = 8, r = 7, k = 5$

Tempos de Execução - Vértices

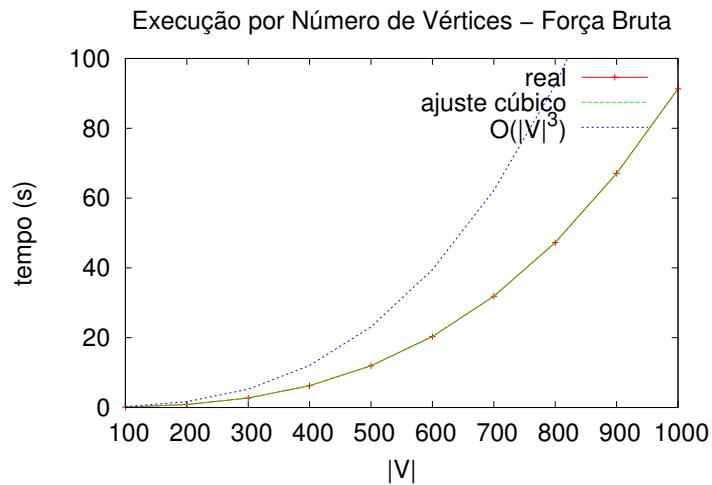


Figure 4: Gráfico de tempos de execução do algoritmo de força bruta para diferentes quantidades de vértices, com $|\mathbb{F}|$ fixo.

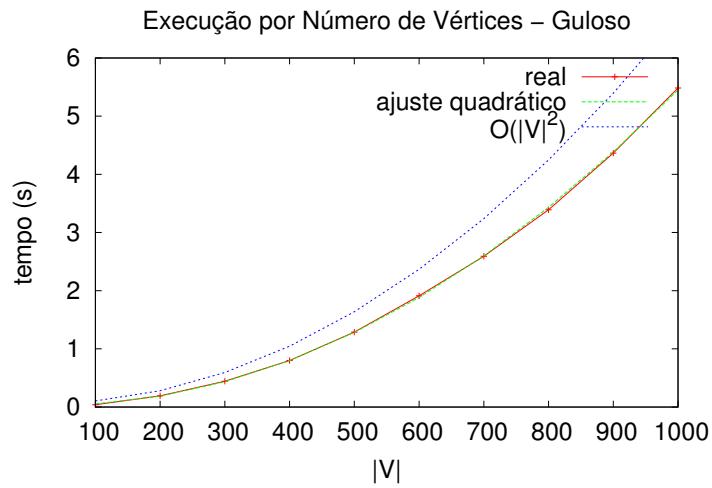


Figure 5: Gráfico de tempos de execução do algoritmo guloso para diferentes quantidades de vértices, com $|F|$ fixo.

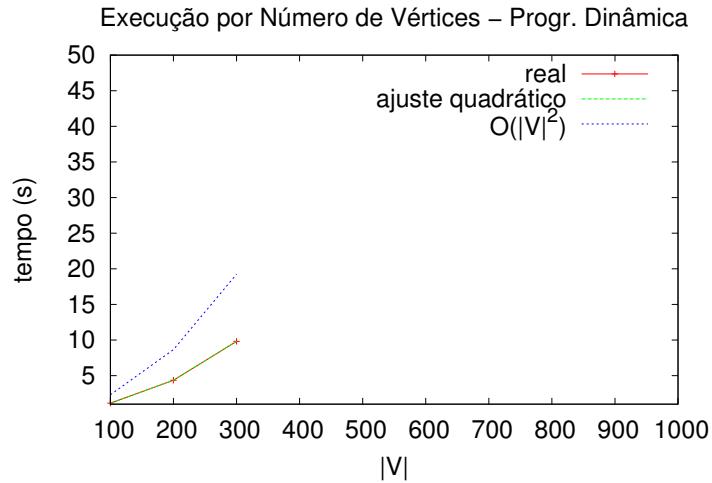


Figure 6: Gráfico de tempos de execução do algoritmo de programação dinâmica para diferentes quantidades de vértices, com $|F|$ fixo. Para valores de $|V| > 300$, o limite de memória de 6GB para execução foi rompido.

Tempos de Execução - Focos

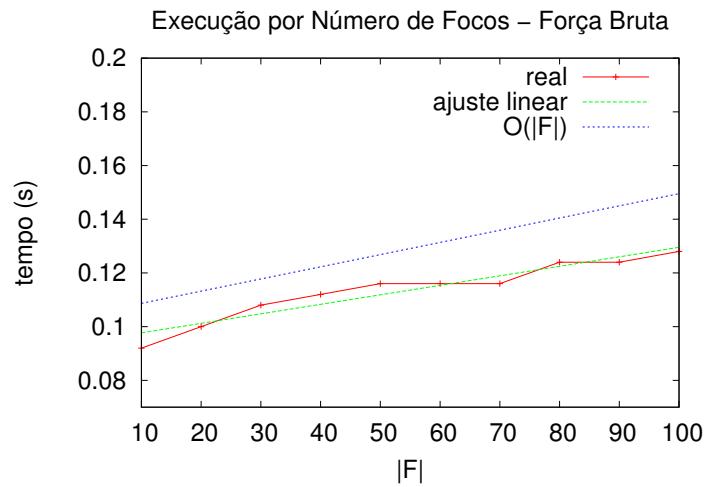


Figure 7: Gráfico de tempos de execução do algoritmo de força bruta para diferentes quantidades de focos, com $|V|$ fixo.

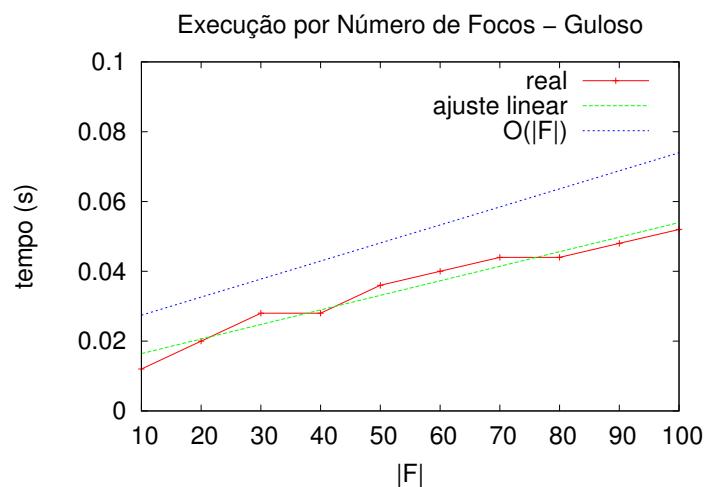


Figure 8: Gráfico de tempos de execução do algoritmo guloso para diferentes quantidades de focos, com $|V|$ fixo.

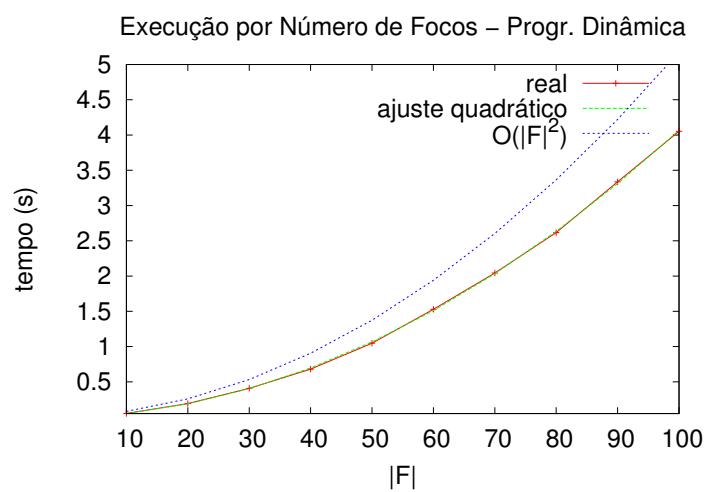


Figure 9: Gráfico de tempos de execução do algoritmo de programação dinâmica para diferentes quantidades de focos, com $|\mathbb{V}|$ fixo.

Exercício 5 - Comparação dos Custos Teóricos com os Tempos de Execução

Força Bruta

No exercício 2, argumentamos que o custo assintótico de execução do algoritmo de força bruta é $\mathcal{O}(|\mathbb{F}||\mathbb{V}|^3)$. Isso implica que, quando $|\mathbb{F}|$ é mantido fixo, o custo deve variar cubicamente com relação ao número de vértices da entrada: $\mathcal{O}(|\mathbb{V}|^3)$. Isso se confirma na figura 4.

Analogamente, mantendo $|\mathbb{V}|$ fixo, o tempo de execução deve variar linearmente com relação ao número de focos da entrada: $\mathcal{O}(|\mathbb{F}|)$. Isso se confirma na figura 7.

Guloso

No exercício 2, argumentamos que o custo assintótico de execução do algoritmo guloso é $\mathcal{O}(|\mathbb{F}||\mathbb{V}|^2)$. Isso implica que, quando $|\mathbb{F}|$ é mantido fixo, o custo deve variar quadraticamente com relação ao número de vértices da entrada: $\mathcal{O}(|\mathbb{V}|^2)$. Isso se confirma na figura 5.

Analogamente, mantendo $|\mathbb{V}|$ fixo, o tempo de execução deve variar linearmente com relação ao número de focos da entrada: $\mathcal{O}(|\mathbb{F}|)$. Isso se confirma na figura 8.

Programação Dinâmica

No exercício 2, argumentamos que o custo assintótico de execução do algoritmo de programação dinâmica é $\mathcal{O}(|\mathbb{F}|^2|\mathbb{V}|^2)$. Isso implica que, quando $|\mathbb{F}|$ é mantido fixo, o custo deve variar quadraticamente com relação ao número de vértices da entrada: $\mathcal{O}(|\mathbb{V}|^2)$. Isso se confirma na figura 6.

Analogamente, mantendo $|\mathbb{V}|$ fixo, o tempo de execução deve variar também quadraticamente com relação ao número de focos da entrada: $\mathcal{O}(|\mathbb{F}|^2)$. Isso se confirma na figura 9.

Projeto e Análise de Algoritmos

Trabalho Prático 2

Mariana Arantes

¹Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

{mariana.arantes}@dcc.ufmg.br

Exercício 1

0.1. Algoritmo Força Bruta

O algoritmo que utiliza o paradigma de busca por força bruta encontra a solução para o problema proposto através da verificação de todas as correntes possíveis de serem construídas com o anel dado. O algoritmo constrói as correntes em ordem crescente de tamanho. Para cada tamanho de corrente, n correntes são geradas, cada uma começando em um vértice diferente. Dessa forma, o algoritmo verifica todas as correntes possíveis e seleciona a menor delas. Em caso de empate, o algoritmo retorna a corrente cuja soma dos identificadores dos vértices seja a menor possível. O método é mostrado no Algoritmo 2.

O primeiro passo do algoritmo consiste em inicializar algumas variáveis auxiliares. A variável *n* contém o número de vértices do anel, a variável *answer* é usada para guardar a solução ótima encontrada até o momento e a variável *sum_* inicialmente contém um limite superior para a soma dos identificadores dos vértices e é usada para realizar o desempate caso exista mais de uma solução ótima. A variável *chain* é um vetor que armazena a conversão dos índices de 0 a n-1 utilizados no código para o valor real dos identificadores dos vértices.

O mapeamento dos índices usados no código para o valor real dos identificadores dos vértices é feito pelo método *find_chain* que é apresentado em Algoritmo 1. O procedimento realizado é simples. O algoritmo seleciona o vértice 1 como o inicial, o nomeia como o vértice de índice 0 e escolhe um dos dois vértices da sua lista de adjacência para prosseguir, esse vértice é então nomeado o vértice de índice 1 usado no código. Após nomear esse vértice, o algoritmo pega um de seus adjacentes e o considera como o vértice de índice 2. O processo prossegue até que o anel se feche. A Figura 1 mostra um exemplo para um anel com 4 vértices. O vetor real contém a sequência do anel com os identificadores dos vértices e o vetor de índices mapeia o vértice 1 para o índice 0, o vértice 3 para o índice 1, o vértice 2 para o índice 2 e por último, o vértice 4 para o índice 3.

Após a inicialização das variáveis e atribuição dos índices, o algoritmo começa a busca pelas cadeias. O algoritmo verifica todas as cadeias de tamanho um, depois todas as de tamanho 2 e termina após verificar todas as de tamanho n. Ao longo da execução, o algoritmo mantém a solução ótima na variável *answer*. Para o exemplo mostrado na Figura 1, as cadeias são buscadas na seguinte ordem: (1), (3), (2), (4), (1, 3), (3, 2), (2, 4), (4, 1), (1, 3, 2), (3, 2, 4), (2, 4, 1), (4, 1, 3), (1, 3, 2, 4), (3, 2, 4, 1), (2, 4, 1, 3) e (4, 1, 3, 2). A análise das últimas três cadeias é redundante, porém como o objetivo era fazer um algoritmo força bruta, elas foram mantidas.

Input: O grafo anel $G(V, E)$.

Output: Um vetor com o mapeamento dos índices para identificadores.

```

chain ← [0]
while chain.size() < |G.V| do
    id_ ← chain[-1]
    adj_list ← G.get_vertex(id_).get_adj_list()
    for vertex in adj_list do
        id_ ← vertex.get_id()
        if id_ not in chain then
            chain.append(id_)
            break
        end
    end
end
return chain

```

Algorithm 1: Mapeia índices para identificadores reais dos vértices.

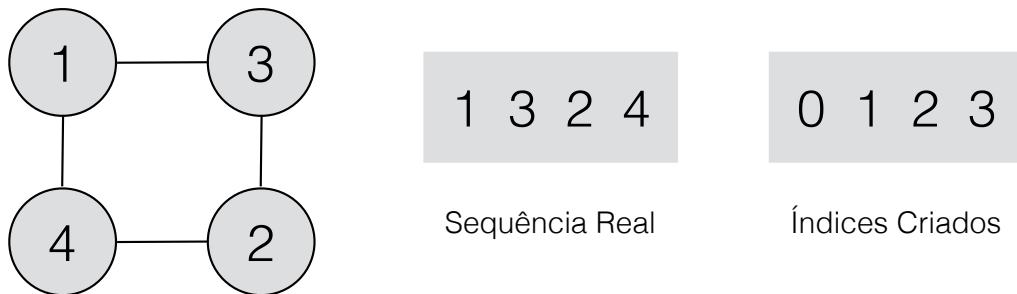


Figura 1. Exemplo de grafo anel com 4 voluntários.

0.2. Algoritmo de Programação Dinâmica

O algoritmo que utiliza o paradigma de programação dinâmica, assim como o de força bruta, verifica as correntes em ordem crescente de tamanho, considerando cada um dos n vértices como ponto inicial. Para cada tamanho de corrente de 1 a $n-1$, n correntes são geradas e a corrente de tamanho n é computada apenas uma vez. Quando o algoritmo encontra uma solução ótima, ele termina de analisar todas as outras correntes de mesmo tamanho da solução ótima encontrada e só então encerra a execução e retorna a solução. Caso exista mais de uma solução ótima, ele escolhe a que contém a menor soma dos identificadores dos vértices. O algoritmo poderia parar assim que encontra a solução ótima, mas a verificação das outras correntes de mesmo tamanho é necessária devido ao requisito de retornar a de menor soma.

Ao verificar os focos cobertos por uma corrente, o algoritmo não faz a união dos focos de cada vértice que pertence a corrente. Um vetor com n conjuntos ($spots_list$) é mantido e a posição i do vetor contém o conjunto de focos coberto pela corrente que começa no vértice de índice i . No inicio os subconjuntos estão vazios e a cada busca por correntes de tamanho $subset_size$, os conjuntos são incrementados para guardar os focos cobertos por correntes que se iniciam no vértice de índice i e tem tamanho $subset_size$.

```

Input: O grafo anel  $G(V, E)$ , um inteiro positivo  $r$  e uma relação
 $\mathbb{R}(v) : V \rightarrow \mathbb{F}$ 
Output: Um subconjunto  $V' \subseteq V$ 
// inicializa variáveis
 $n \leftarrow |G.V|$ 
 $chain \leftarrow \text{find\_chain}(G)$ 
 $answer \leftarrow \text{set}()$ 
 $sum\_ \leftarrow n * (n + 1)$ 
// verifica todos os tamanhos de cadeias possíveis
for  $subset\_size \leftarrow 1$  to  $n$  do
    // escolhe o vértice inicial da cadeia
    for  $start \leftarrow 1$  to  $n - 1$  do
         $subset \leftarrow \text{set}()$ 
         $ss\_spots \leftarrow \text{set}()$ 
        // encontra cadeia
        for  $v \leftarrow start$  to  $start + subset\_size - 1$  do
             $id\_ \leftarrow chain[v \% n]$ 
             $subset.add(id\_)$ 
             $vertex \leftarrow G.\text{get\_vertex}(id\_)$ 
             $ss\_spots \leftarrow ss\_spots.\text{union}(vertex.\text{get\_spots}())$ 
        end
        // verifica se cadeia encontrada cobre todos os focos
        if  $ss\_spots.size() == r$  then
            if  $sum(subset) < sum\_ \text{ and } subset.size() \leq answer.size()$  then
                 $answer \leftarrow subset$ 
                 $sum\_ \leftarrow sum(subset)$ 
            end
        end
    end
end
return  $answer$ 

```

Algorithm 2: Algoritmo que utiliza o paradigma de força bruta.

Dessa forma, ao computar a corrente de tamanho $subset_size$, a corrente de tamanho $subset_size - 1$ com início no mesmo vértice já foi computada. Para obter o conjunto de focos, basta realizar uma união de conjuntos dos focos do novo vértice que foi adicionado com os focos da corrente menor. O método é mostrado em Algoritmo 3.

A Figura 2 mostra o exemplo de execução do algoritmo para o grafo anel apresentado na Figura 1. A tabela da esquerda mostra os focos cobertos por cada vértice e a tabela da direita mostra o conteúdo dos vetores $subset_list$ e $spots_list$ para cada tamanho de corrente $subset_size$. O algoritmo encontra uma solução de tamanho 3, termina de verificar todas as cadeias de tamanho 3 e encerra a execução. A solução ótima está em negrito na tabela.

```

Input: O grafo anel  $G(V, E)$ , um inteiro positivo  $r$  e uma relação
 $\mathbb{R}(v) : V \rightarrow \mathbb{F}$ 
Output: Um subconjunto  $V' \subseteq V$ 
// inicializa variáveis
 $n, chain, answer, sum_ \leftarrow start\_variables()$ 
// vetores utilizados para armazenar os focos cobertos por cada corrente
subset_list = []
spots_list = []
// inicializa vetores
for  $i \leftarrow 0$  to  $n - 1$  do
| subset_list.append(set())
| spots_list.append(set())
end
// verifica todos os tamanhos de cadeias possíveis
for subset_size  $\leftarrow 1$  to  $n$  do
| // se ja tem uma solução, não precisa procurar por soluções maiores if
| answer.size()  $> 0$  then
| | break
| end
| // para cada ponto inicial
| for  $i \leftarrow 0$  to  $n - 1$  do
| | vertex  $\leftarrow G.get\_vertex(chain[(i + subset\_size - 1) \% n])$ 
| | id_  $\leftarrow vertex.get\_id()$ 
| | ss_spots  $\leftarrow vertex.get\_spots()$ 
| | subset_list[i].add(id_)
| | // usa focos da cadeia menor para computar para essa cadeia
| | spots_list[i]  $\leftarrow spots\_list[i].union(ss\_spots)$ 
| | // verifica se cadeia cobre todos os focos
| | if ss_spots.size()  $= r$  then
| | | // mantém solução com a menor soma
| | | if sum(subset)  $<$  sum_ then
| | | | answer  $\leftarrow subset$ 
| | | | sum_  $\leftarrow sum(subset)$ 
| | | end
| | | // interrompe para computar cadeia de tamanho n apenas uma vez
| | | if subset_size == n then
| | | | break
| | | end
| | end
| end
| end
return answer

```

Algorithm 3: Algoritmo que utiliza o paradigma de programação dinâmica.

0.3. Algoritmo Guloso

O algoritmo que utiliza o paradigma guloso também verifica as correntes em ordem crescente de tamanho, considerando cada um dos vértices como o vértice inicial. No

Vértice	Focos	subset_size	subset_list	spots_list
1	1, 2	1	{1}, {3}, {2}, {4}	{1,2}, {3,4}, {5,6}, {1,3}
2	5, 6	2	{1,3}, {3,2}, {2,4}, {4,1}	{1,2,3,4}, {3,4,5,6}, {1,3,5,6}, {1,2,3}
3	3, 4			
4	1, 3	3	{1,3,2}, {3,2,4}, {2,4,1}, {4,1,3}	{1,2,3,4,5,6}, {1,3,4,5,6}, {1,2,3,5,6}, {1,2,3,4}

Figura 2. Exemplo de execução do algoritmo dinâmico para o grafo anel da Figura 1.

entanto, para cada tamanho de corrente, a ordem em que elas são analisadas não é sequencial. A escolha da próxima corrente a ser verificada é feita por uma estratégia gulosa. Para fazer a escolha, primeiro o algoritmo computa um *score* para cada foco e um *score* para cada vértice.

O *score* de um foco é definido como o número de vértices do grafo anel que cobrem aquele foco e o *score* de um vértice é definido como a soma dos *scores* dos focos cobertos pelo vértice. A idéia é que um vértice com *score* baixo tem uma probabilidade maior de estar na solução ótima, uma vez que ele cobre focos que são cobertos por poucos vértices. Se um vértice tem *score* dois por exemplo, ele obrigatoriamente vai fazer parte da solução ótima, pois ele é o único vértice a cobrir os dois focos que ele cobre. Seguindo esse raciocínio, o algoritmo vai analisar as cadeias por ordem de tamanho e para tamanho *subset_size*, ele vai analisar primeiro todas as cadeias de tamanho *subset_size* que contém o vértice de menor *score*, depois todas as cadeias de tamanho *subset_size* que contém o segundo vértice de menor *score* até que todos os vértices tenham sido utilizados. Dessa forma, pode ser que o algoritmo encontre a solução ótima mais rapidamente.

Com essa estratégia, várias cadeias são verificadas mais de uma vez. Para evitar a redundância na computação de focos, uma matrix $M_{n,n}$ é mantida. As linhas da matrix representam o ponto inicial da corrente e as colunas representam o tamanho da corrente. O elemento $m_{i,j}$ da matrix representa os focos cobertos pela corrente de tamanho j com início no vértice i . Dessa forma, ao computar correntes repetidas, não é necessário recalcular os focos.

Assim como no algoritmo de programação dinâmica, ao encontrar uma solução ótima, o algoritmo termina de verificar todas as cadeias do mesmo tamanho da que já foi encontrada e depois encerra a execução, escolhendo, em caso de empate, a solução com menor soma dos identificadores dos vértices. Se o requisito de retornar a solução ótima de menor soma não for considerado, o algoritmo pode apresentar uma performance melhor que o algoritmo dinâmico, porém com esse requisito, os dois algoritmos vão computar o mesmo número de correntes, uma vez que eles precisam verificar todas com mesmo tamanho da solução já encontrada antes de encerrar a execução. O método é apresentado em Algoritmo 4.

Exercício 2

Análise de Complexidade do Algoritmo Força Bruta

Tempo

O algoritmo analisa n^2 correntes. Para cada corrente, realiza no máximo n operações de adição a conjunto. Como cada operação de adição pode ser implementada em tempo constante (utilizando vetor de bits por exemplo), a complexidade total é $O(n^3)$.

Espaço

O algoritmo utiliza espaço $O(n)$ para armazenar o grafo, $O(n + r) = O(n)$ para armazenar os focos e $O(r)$ para guardar a solução. Logo, a complexidade de espaço total é $O(n)$.

Análise de Complexidade do Algoritmo de Programação Dinâmica

Tempo

O algoritmo analisa no máximo n^2 correntes. Para cada corrente, gasta um tempo constante para analisar os focos, logo a complexidade total é $O(n^2)$.

Espaço

O algoritmo faz uso de um vetor para guardar o conjunto de focos cobertos. São n conjuntos de tamanho no máximo r. A complexidade de espaço total é $O(nr)$.

Análise de Complexidade do Algoritmo Guloso

Tempo

O algoritmo tem complexidade de tempo igual a complexidade do algoritmo de programação dinâmica. Ele preenche no máximo uma matrix de tamanho n^2 , para as outras correntes, apenas usa a entrada já existente na matriz. A complexidade total é $O(n^2)$.

Espaço

O algoritmo mantém uma matriz quadrada de dimensão n e cada posição dessa matriz guarda um conjunto de focos. Logo a complexidade de espaço é $O(n^2r)$.

Exercício 4

Um gerador de grafos foi implementado e usado para gerar dados de entrada para o problema. Uma distribuição uniforme foi utilizada para atribuir focos a vértices, sempre garantindo que todo foco seja coberto por pelo menos um vértice. Ao todo, foram gerados grafos anéis com 100, 200, 300, 400 e 500 vértices e cada um deles com número de focos

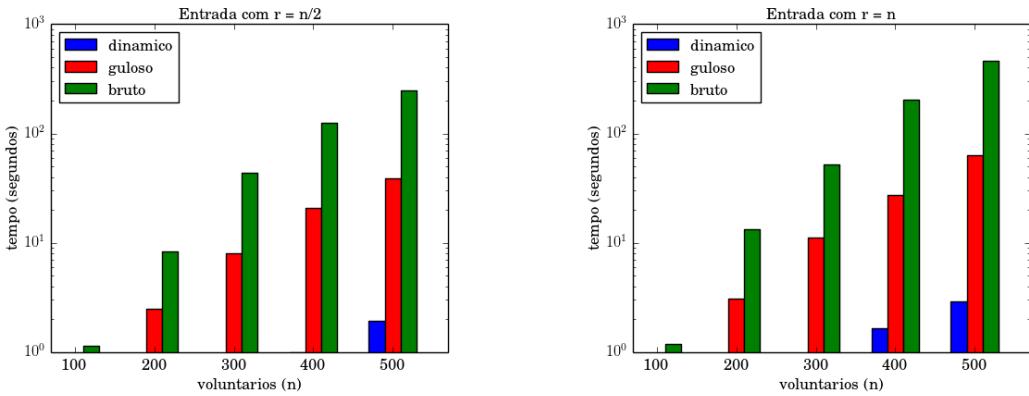


Figura 3. Tempo de execução para diferente valores de n e r.

igual a metade do número de vértices ou igual ao número de vértices. A Figura 3 mostra o tempo de execução (em segundos), para cada um dos algoritmos propostos quando executados com as entradas geradas.

Exercício 5

Os gráficos apresentados na Figura 3 mostram que o algoritmo que utiliza o paradigma de força bruta é o menos eficiente deles, o que condiz com a análise de complexidade, pois esse algoritmo tem complexidade $O(n^3)$, enquanto os outros dois algoritmos tem complexidade $O(n^2)$. O algoritmo que usa o paradigma de programação dinâmica apresenta o melhor resultado, pois o algoritmo evita o recálculo de muitos conjuntos de focos, não verifica cadeias repetidas e encerra a sua execução após encontrar uma solução ótima e verificar todas as outras cadeias de mesmo tamanho da solução encontrada. O algoritmo guloso, embora apresente a mesma complexidade teórica do algoritmo de programação dinâmica, não se mostrou tão eficiente na prática, embora seja melhor que o algoritmo de força bruta. Esse comportamento pode ser explicado devido aos processamentos adicionais que são necessários no algoritmo guloso para obter o *score* de cada vértice, além da possível verificação da mesma corrente mais de uma vez.

O gráfico da esquerda mostra o tempo de execução de cada algoritmo para o número de focos igual a metade do número de vértices e o grafo da direita mostra o tempo de execução para o número de focos igual ao número de vértices. Quanto menos focos o problema tiver, mais rápido os algoritmos executam, pois o número de vértices necessários para cobrir todos os focos é menor. A diferença no tempo de execução é maior para os algoritmos dinâmico e guloso, pois eles encerram a execução assim que encontram uma solução ótima, enquanto o algoritmo de força bruta verifica todas as cadeias, independente da solução já ter sido encontrada.

Input: O grafo anel $G(V, E)$, um inteiro positivo r e uma relação

$$\mathbb{R}(v) : \mathbb{V} \rightarrow \mathbb{F}$$

Output: Um subconjunto $V' \subseteq V$

// inicializa variáveis

$n, chain, answer, sum_ \leftarrow start_variables()$

// encontra o score de cada vértice

$vertex_score \leftarrow compute_score()$

// inicializa matriz M

$M \leftarrow dict()$

for $start \leftarrow 0$ **to** $n - 1$ **do**

$| M[(start, 1)] \leftarrow G.get_vertex(chain[start]).get_spots()$

end

// verifica todos os tamanhos de cadeias possíveis

for $subset_size \leftarrow 1$ **to** n **do**

 // se ja tem uma solução, não precisa procurar por soluções maiores

if $answer.size() > 0$ **then**

 | break

end

 // para cada vértice em ordem crescente de score

for $id_ in vertex_score$ **in increasing order do**

 // indices para andar no anel

$index \leftarrow id_$

$start \leftarrow index - subset_size + 1$

while $start \leq index$ **do**

$| subset \leftarrow set()$

 // constrói corrente

for $i \leftarrow 0$ **to** $subset_size - 1$ **do**

 | $| subset.add(chain[(start + 1)\%n])$

end

 // utiliza matriz para computar focos

$ss_spots \leftarrow get_spots()$

 // verifica se cadeia cobre todos os focos

if $ss_spots.size() == r$ **then**

 // mantém solução com a menor soma

if $sum(subset) < sum_ and subset.size() \leq answer.size()$

then

 | $| answer \leftarrow subset$

 | $| sum_ \leftarrow sum(subset)$

end

end

$start \leftarrow start + 1$

end

end

return $answer$

Algorithm 4: Algoritmo que utiliza o paradigma guloso.

UFMG – PPGCC/DCC – Projeto e Análise de Algoritmos (PAA) – 1º Semestre de 2016
Paradigmas de Projeto de Algoritmos – Trabalho Prático: ZikaZeroAnelDual

Aluno: Tiago Amador Coelho

Exercício 1:

1 – Busca por Força-Bruta

n = # vertices

Para i = 1 até n

cria subgrafo

subgrafo recebe vertice i

Para j = i até n-1

Se subgrafo contem todos os focos

Compara com o melhor subgrafo até o momento e se for melhor, armazena-o

break

Acrescento vertice adjacente ao vertice i ao subgrafo

Fim_Para

Fim_Para

2 – Programação Dinâmica

n = # vertices

tam_Melhor_Subgrafo = n

Para i = 1 até n

cria subgrafo

subgrafo recebe vertice i

Enquanto Verdade

Se subgrafo contem todos os focos

Compara com o Melhor_Subgrafo até o momento e se for melhor, armazena-o

break

Se tam_Melhor_Subgrafo <= tam_Subgrafo

break

Fim_Se

Acrescento vertice adjacente esquerdo ao subgrafo

Fim_Enquanto

Fim_Para

3 – Algoritmo Guloso

```
n = # vertices
Para i = 1 até n
    cria subgrafo
    subgrafo recebe vertice i

    Para j = i até n-1
        Se subgrafo contem todos os focos
            Compara com o melhor subgrafo até o momento e se for melhor, armazena-o
            break

        // ganho é a aresta que acrescenta mais focos não visitados ao subgrafo
        Calcula o ganho dos vertices adjacentes ao subgrafo
        Acrescenta vertice adjacente com o maior ganho ao vertice i ao subgrafo

    Fim_Para
Fim_Para
```

Exercício 2:

1 – Busca por Força-Bruta
Complexidade Tempo: $O(n^2)$
Complexidade Espaço: $O(n^2)$ pois utiliza de matrix

2 – Programação Dinâmica
Complexidade Tempo: $O(n^2)$
Complexidade Espaço: $O(n)$ utiliza de vetor para representar o grafo

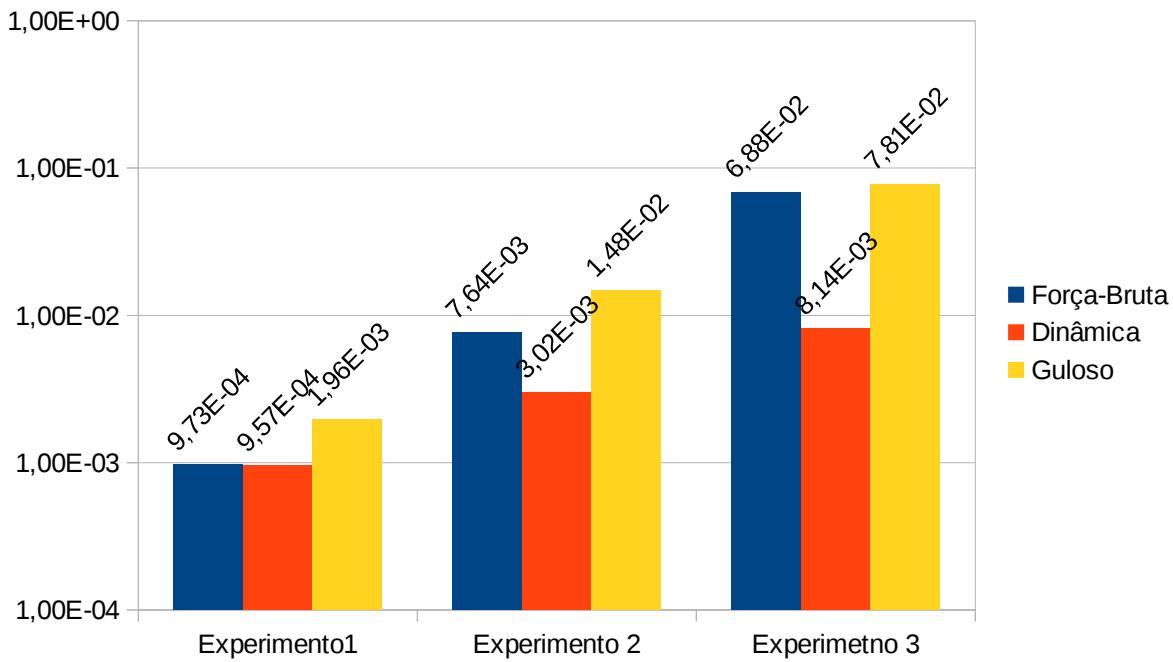
3 – Algoritmo Guloso
Complexidade Tempo: $O(n^2)$
Complexidade Espaço: $O(n)$ utiliza de vetor para representar o grafo

Exercício 3:

Em anexo os códigos

Exercício 4:

Entradas			Força-Bruta	Dinâmica	Guloso
<i>n</i>	<i>m</i>	<i>r</i>	Tempo (s)	Tempo (s)	Tempo (s)
7	7	6	0.000972986221313	0.000957012176514	0.00196313858032
23	23	6	0.0076367855072	0.00301885604858	0.014760017395
50	50	10	0.0687928199768	0.00814294815063	0.078097820282



Exercício 5:

Os algoritmos se comportaram como o esperado pela sua ordem de complexidade, $O(n^2)$, pela projeção pode-se observar que o algoritmo de força-bruta em um experimento com um maior número de n irá levar mais tempo de execução que os outros dois algoritmos. Isso se deve pela quantidade de comparações que a técnica utiliza na sua execução.

O paradigma de programação dinâmica é aquele que tende a fazer o menor número de operações, já que ela troca processamento por armazenamento.

UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROJETO E ANÁLISE DE ALGORITMOS
1º SEMESTRE DE 2016

2º Trabalho Prático de Projeto e Análise de Algoritmos

Aluno: Geanderson Esteves dos Santos
geanderson@dcc.ufmg.br

Professores: Luiz Chaimowicz, Sebastián Urrutia, Wagner Meira Jr. e Jussara M. Almeida

Exercício 1. Proponha algoritmos de Complexidade Assintótica Polinomial com o Tamanho da entrada (n , m e r) para resolver o Problema ZikaZeroAnelDual usando os seguintes paradigmas.

1. Busca por força bruta.

Algoritmo 1: Força Bruta

Data: Todos os vértices e focos armazenados na classe *Graph*

Result: O menor conjunto de vértices conectados que atendem todos os focos inicialização;

escolher vértice no índice 1;

while nem todos os vértices foram checados **do**

| checar todos os adjacentes do vértice corrente armazenando os focos atendidos até que o número de focos verificados seja igual ao resultado;

| **if** focos verificados são iguais ao resultado **then**

| | armazena vértices que resolvem o problema;

| | break;

| **else**

| | verifica próximo vértice e seus adjacentes;

| **end if**

| **end while**

2. Programação Dinâmica

Algoritmo 2: Programação Dinâmica (criação da datatable)

Data: Todos os vértices armazenados na classe *Graph*

Result: Um *datatable* com os focos que o mínimo número de vértices atendem inicialização;

escolher vértice no índice 1;

while nem todos os vértices foram checados **do**

| checar o menor número de vértices e seus respectivos focos que eles podem atender;

| gravar a informação no datatable;

| **end while**

Algoritmo 3: Programação Dinâmica

Data: Todos os vértices e focos armazenados na classe *Graph*
Result: O menor conjunto de vértices conectados que atendem todos os focos
inicialização;
percorrer cada elemento do conjunto de vértices com menor conflito gerado pelo
algoritmo anterior;
while nem todos os vértices foram checados **do**
 | checar todos os adjacentes do vértice corrente armazenados no datatable;
 | **if** focos verificados são iguais ao resultado **then**
 | | armazena vértices que resolvem o problema;
 | | break;
 | **else**
 | | verifica próximo vértice e seus adjacentes;
 | **end if**
end while

3. Algoritmo Guloso

Algoritmo 4: Algoritmo Guloso (Verificação de conflito dos focos)

Data: Todos os vértices e focos armazenados na classe *Graph*
Result: Conjunto de vértices em ordem decrescente baseado no índice de conflito
com outros vértices
inicialização;
escolher vértice no índice 1;
while nem todos os focos foram checados **do**
 | percorrer todos os vértices somando a quantidade de aparição de cada foco;
end while
while nem todos os vértices foram checados **do**
 | percorrer cada vértice checando a soma de aparição dos seus focos;
end while
ordernar o conjunto de vértices em ordem decrescente baseado no conflito com
outros vértices;

Algoritmo 5: Algoritmo Guloso

Data: Todos os vértices e focos armazenados na classe *Graph*

Result: O menor conjunto de vértices conectados que atendem todos os focos inicializaçāo;

percorrer cada elemento do conjunto de vértices com menor conflito gerado pelo algoritmo anterior;

while nem todos os vértices foram checados **do**

| checar todos os adjacentes do vértice corrente armazenando os focos atendidos até que o número de focos verificados seja igual ao resultado;

| **if** focos verificados são iguais ao resultado **then**

| | armazena vértices que resolvem o problema;
| | break;

| **else**

| | verifica próximo vértice e seus adjacentes;

| **end if**

end while

Exercício 2. Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) dos algoritmos propostos no Exercício 1.

(a) Busca por força bruta:

i. Temporal:

$$\begin{aligned} BruteForce &= O(n) \times O(2n) \times O(n) \\ &= O(2n^3) \\ &= O(n^3) \end{aligned} \tag{1}$$

ii. Espacial:

$$\begin{aligned} SpaceBF &= O(n) \times O(r) \\ &= O(n \times r) \end{aligned} \tag{2}$$

(b) Programação Dinâmica:

i. Temporal:

$$\begin{aligned} Dynamic &= O(n) \times O(2n) \\ &= O(2n^2) \\ &= O(n^2) \end{aligned} \tag{3}$$

ii. Espacial:

$$\begin{aligned} SpaceDynamic &= O(n \times r) \times O(n) \times O(r) \\ &= O(n^2 \times r^2) \end{aligned} \tag{4}$$

É importante ressaltar que o algoritmo dinâmico armazena uma tabela de custo N que pode interferir na eficiéncia do algoritmo para um valor grande de N.

(c) Algoritmo Guloso:

i. Temporal:

$$\begin{aligned} Greedy &= O(n^2) \times O(2n) \\ &= O(2n^3) \\ &= O(n^3) \end{aligned} \tag{5}$$

É importante ressaltar que a análise do guloso é amortizada, pois o algoritmo tenta escolher o resultado ótimo. Ao contrário do força bruta que sempre começa a execução no vértice 1, a proposta gulosa olha a quantidade de conflitos e começa a executar no vértice que possui a menor quantidade de conflitos no grafo.

ii. Espacial:

$$\begin{aligned} SpaceGreedy &= O(n) \times O(n) \times O(r) \\ &= O(n^2 \times r) \end{aligned} \tag{6}$$

O custo acrescentado de N no guloso comparado com o força bruta, ocorre devido a criação do vetor de conflitos. Este vetor ordena os vértices em função dos que têm menos conflito no grafo.

Exercício 3. Implemente os algoritmos propostos no Exercício 1 na linguagem de programação C, C++, Java ou Python.

Os algoritmos foram implementados na linguagem de programação Java e estão armazenados na pasta *src*. Os métodos de compilação e execução seguem os propostos pelo enunciado do trabalho. Diversas instâncias interessantes foram implementadas para testar a eficácia dos três algoritmos propostos.

Exercício 4. Execute testes da implementação do Exercício 3, propondo instâncias interessantes para o Problema ZikaZeroAnelDual. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n , m e r) comparando os três algoritmos implementados.

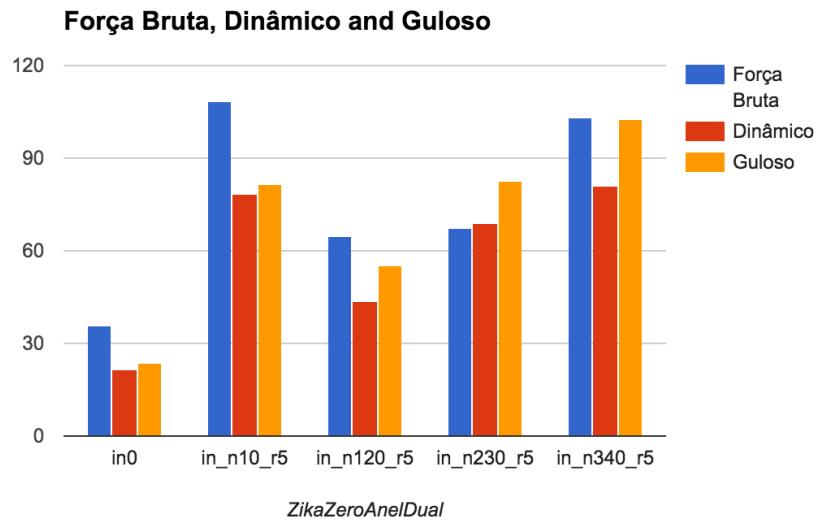


Figura 1: Testes com instâncias pequenas

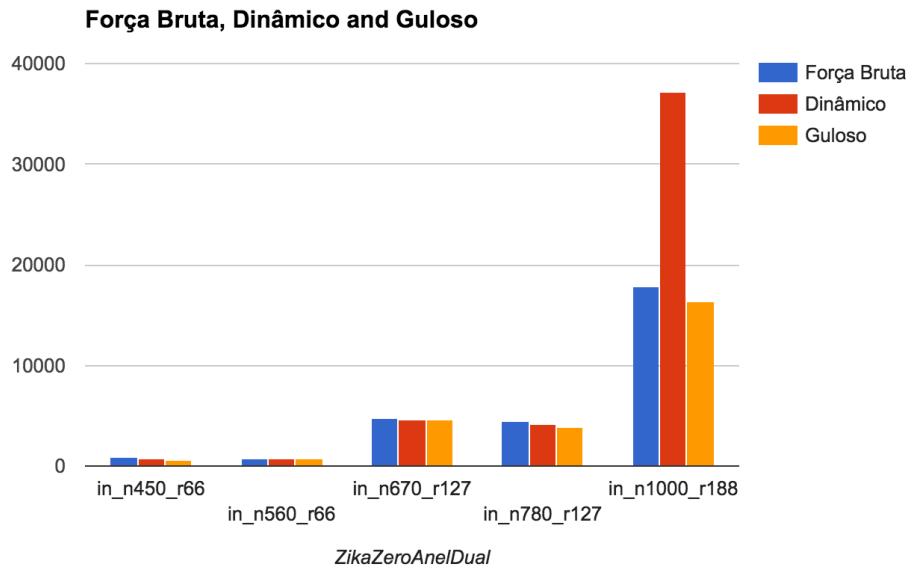


Figura 2: Testes com instâncias grandes

Exercício 5 - Compare a análise e a execução respectivamente, Exercícios 2 e 4. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

A partir da realização dos testes com diferentes instâncias, variando o número de voluntários e focos atendidos por eles, pode-se perceber que os algoritmos guloso e dinâmico foram mais eficientes em encontrar uma solução para o problema do que o força bruta (Figura 1 e 2). Entretanto, este ganho não pode ser considerado alto de nenhuma maneira. Duas hipóteses podem ser argumentadas para justificar o baixo ganho de tempo dos dois algoritmos em relação ao força bruta:

- A técnica para escolha do ótimo no guloso não foi totalmente eficiente, ou seja, a quantidade de conflitos entre os vértices do grafo. Em outras palavras, todas as aparições de focos atendidos pelos vértices do grafo seriam somados, e ao final, um vetor ordenado determinaria quais vértices seriam escolhidos primeiro. O vértice de menor número de conflitos não era necessariamente o resultado ótimo do algoritmo. Em alguns testes realizados pelo autor deste trabalho, notou-se que o resultado ótimo ficou na terceira ou quarta posição no vetor de vértices que o algoritmo computaria primeiro. Infelizmente, tais testes só podem ser computados quando o algoritmo está pronto. Entretanto, é interessante notar que o guloso foi em todos os testes realizados mais rápido que o força bruta, e que para instâncias grandes o guloso mostrou-se, até mesmo, melhor que o algoritmo de programação dinâmico.
- O algoritmo de programação dinâmica apresenta um custo espacial desnecessário para instâncias muito grandes, isso porque, o algoritmo armazena uma tabela de custo N que pode interferir na eficiência do algoritmo para um valor grande de N . Ou seja, a sub-estrutura ótima que foi desenvolvida para evitar que o problema do ZikaZeroAnelDual calcule resultados desnecessários que já foram calculados anteriormente, mostrou-se mais interessantes para um valor pequeno de N , tornando-se menos interessantes à medida que N cresce no tempo. Isso acontece, sobretudo pelas características do problema ZikaZeroAnelDual, que não é necessariamente, um problema de difícil resolução, ou seja, o enunciado do próprio trabalho impede que o problema seja NP-Completo.

Outra característica interessante do problema do ZikaZeroAnelDual, e que deve ser relatada, é que o problema não é NP-Completo. As características intrínsecas do problema, como por exemplo, todos os vértices atendem o mesmo número de voluntários e o fato dos voluntários formarem um anel no grafo, impediram que o problema crescesse exponencialmente. Portanto, o ganho dos paradigmas de programação dinâmico e guloso, não são percebidos da mesma forma que seriam quando se tratando de um problema NP-Completo que cresce exponencialmente. Especialmente o algoritmo dinâmico, pois o mesmo cria uma tabela de tamanho N que aumenta o custo espacial do algoritmo com o tempo, e que torna sua execução com um número elevado de vértices (e.g., 1000 vértices), menos interessante do que o algoritmo guloso.

UFMG/ICEx/DCC
Pós-Graduação em Ciência da Computação
Projeto e Análise de Algoritmos
Trabalho Prático 2 - Paradigmas

Jessica Sena de Souza

Problema ZikaZeroAnelDual

Dados um grafo anel $\mathcal{G}_o(\mathbb{V}, \mathbb{A})$, em que \mathbb{V} é o conjunto de n voluntários e \mathbb{A} é o conjunto de $m = |\mathbb{V}|$ laços de amizade, um conjunto \mathbb{F} dos r focos de reprodução do mosquito, e, uma relação $\mathcal{R}(v) : \mathbb{V} \rightarrow \mathbb{F}$, definida para cada $v \in \mathbb{V}$, explicitando os focos de reprodução aos quais o voluntário v tem acesso, sendo que $|\mathcal{R}(v)| = 2, \forall v \in \mathbb{V}$. 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 \mathcal{G}_o é conexo.

Exercício 1

Busca por Força-bruta

No algoritmo de força bruta o grafo foi implementado como uma lista de adjacências e os focos são armazenados em uma vetor onde o índice representa o voluntário e o conteúdo da posição (dois elementos inteiros), são os focos que o voluntário acessa. A restrição do grafo para esse problema, onde o grafo \mathcal{G}_o é um grafo anel, possibilita a geração de subconjuntos em tempo polinomial, e portanto possibilita a busca por força bruta também em tempo polinomial. Sendo assim o algoritmo proposto gera todas as possibilidades de subconjuntos dentro da restrição de grafo anel, e para cada geração, testa se os voluntários desse subconjunto cobrem todos os focos. Cada subconjunto que cobre todos os focos é armazenado e no final é escolhido o subconjunto composto pelo menor número de voluntários. Caso haja mais de um conjunto com o menor número de voluntários, o desempate é feito escolhendo o subconjunto que foi gerado primeiro. O Algoritmo 1 expressa a implementação descrita anteriormente. É possível ver uma ilustração da execução do Algoritmo 1 nas Figuras 1 e 2.

Programação Dinâmica

A programação dinâmica foi elaborada em cima da sobreposição de problemas encontrados na geração de subconjuntos do grafo \mathcal{G}_o . A Figura 1 mostra a instância do grafo \mathcal{G}_o dada na especificação do problema. Os vértices estão simbolizados com cores diferentes para auxiliar a visualização da sobreposição de problemas. A Figura 2 mostra parte do algoritmo de busca por força bruta. Como é possível notar, a propriedade do grafo anel proporciona uma busca sequencial pelo grafo, o que gera uma sobreposição de problemas e um reaproveitamento de cálculos. Os retângulos coloridos mostram que todos os vértices, exceto o vértice inicial, aparecem na mesma ordem na interação subsequente. Desta maneira, é possível reaproveitar todo esse cálculo apenas fazendo o ajuste de retirar a colaboração do primeiro vértice dos resultados de cada subconjunto da interação anterior e posteriormente acrescentar essa colaboração no subconjunto que contém todos os vértices na interação atual.

Algorithm 1 Força Bruta Gera todos os subconjuntos \mathbb{V}' para um grafo $\mathcal{G}_o(\mathbb{V}, \mathbb{A})$, verifica quais geram uma solução para o problema *ZikaZeroAnelDual* e retorna a menor solução.

Require: Grafo $\mathcal{G}_o(\mathbb{V}, \mathbb{A})$, Conjunto de inteiros $SubSet$, Conjunto de conjuntos $Solucoes$

```
1: function FORÇA BRUTA( $\mathcal{G}_o(\mathbb{V}, \mathbb{A})$ ,  $SubSet$ ,  $Solucoes$ )
2:   for all Vértice  $v \in \mathbb{V}$  do
3:      $SubSet \leftarrow v$ 
4:     for all Vértice  $u \in \mathbb{V} \setminus v$  do            $\triangleright$  percorre os vértices de maneira sequencial a partir de  $v$ 
5:       if Todos os  $r$  focos são cobertos por  $SubSet$  then
6:          $Solucoes \leftarrow SubSet$ 
7:          $SubSet \leftarrow u$ 
8:       Remove todos elementos de  $SubSet$ 
9:     return menor elemento de  $Solucoes$ 
```

Algorithm 2 Programação Dinâmica

Require: Tabela de memorização M , Vetor C com a relação $\mathcal{R}(v) : \mathbb{V} \rightarrow F$, Grafo $\mathcal{G}_o(\mathbb{V}, \mathbb{A})$

```
1: function FORÇA BRUTA( $\mathcal{G}_o(\mathbb{V}, \mathbb{A})$ )
2:   for all Vértice  $v \in \mathbb{V}$  do
3:      $M[i][i] = C[i]$ 
4:     for all Vértice  $u \in \mathbb{V} \setminus v$  do            $\triangleright$  percorre os vértices de maneira sequencial a partir de  $v$ 
5:       if  $i == 0$  then
6:          $M[i][j] = M[i][j - 1] \cup C[i]$ 
7:       else
8:         if  $j == (i - 1)$  then
9:            $M[i][j] = M[i][j - 1] \cup C[i]$ 
10:        else
11:           $M[i][j] = M[i - 1][j] \setminus M[i - 1][i - 1]$ 
12:        if  $|M[i][j] \cap F| == r$  then
13:           $Solucoes \leftarrow SubSet$ 
14:        return menor elemento de  $Solucoes$ 
```

Equação de Recorrência

Na Equação 1, i e j são as posições da tabela de memorização M e também são o vértice inicial daquela interação (i) e o vértice atual do percurso no grafo (j). Assim, $M[i][j]$ é a posição atual que representa a cobertura de focos até o momento da combinação dos vértices de i até j para formar uma solução. $M[i][j - 1]$ é o vértice anterior a esse no grafo \mathcal{G}_o , que contem a cobertura de focos até aquele momento na interação iniciada no foco i . $M[i][i]$ é a posição inicial da interação, onde é guardado somente a cobertura daquele vértice uma vez que só tem ele na solução. $M[i - 1][i]$ é a acumulação do vértice atual na interação anterior (iniciada pelo vértice $i-1$). C é um vetor com a relação $\mathcal{R}(v) : \mathbb{V} \rightarrow F$, onde cada índice é um vértice e o conteúdo são os focos cobertos por esse vértice. O Algoritmo 2 implementa a equação de recorrência.

$$\forall i \in \mathcal{G}_o$$

$$M[i][j] = \begin{cases} M[i][j - 1] \cup C[i] & \text{if } i = 0 \text{ ou } j = (i - 1) \\ M[i - 1][j] \setminus M[i - 1][i - 1] & \text{otherwise} \end{cases} \quad (1)$$

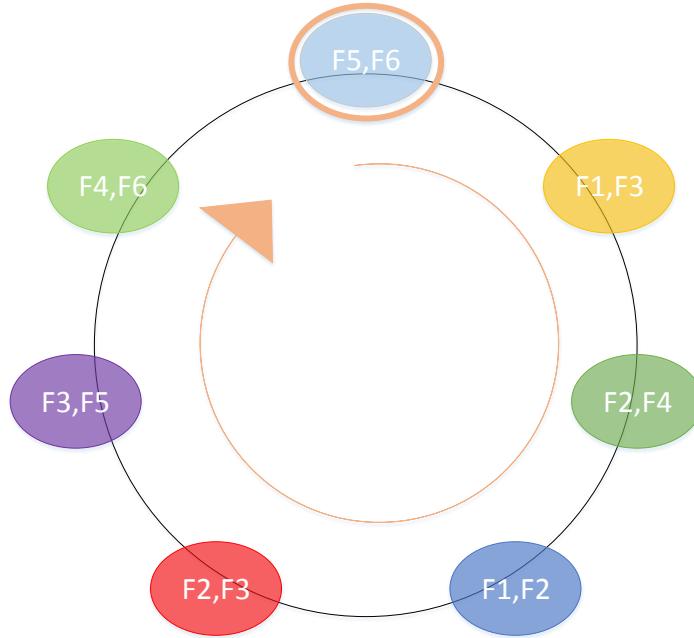


Figure 1: Instância do grafo \mathcal{G}_o simbolizando os vértices com cores diferentes.

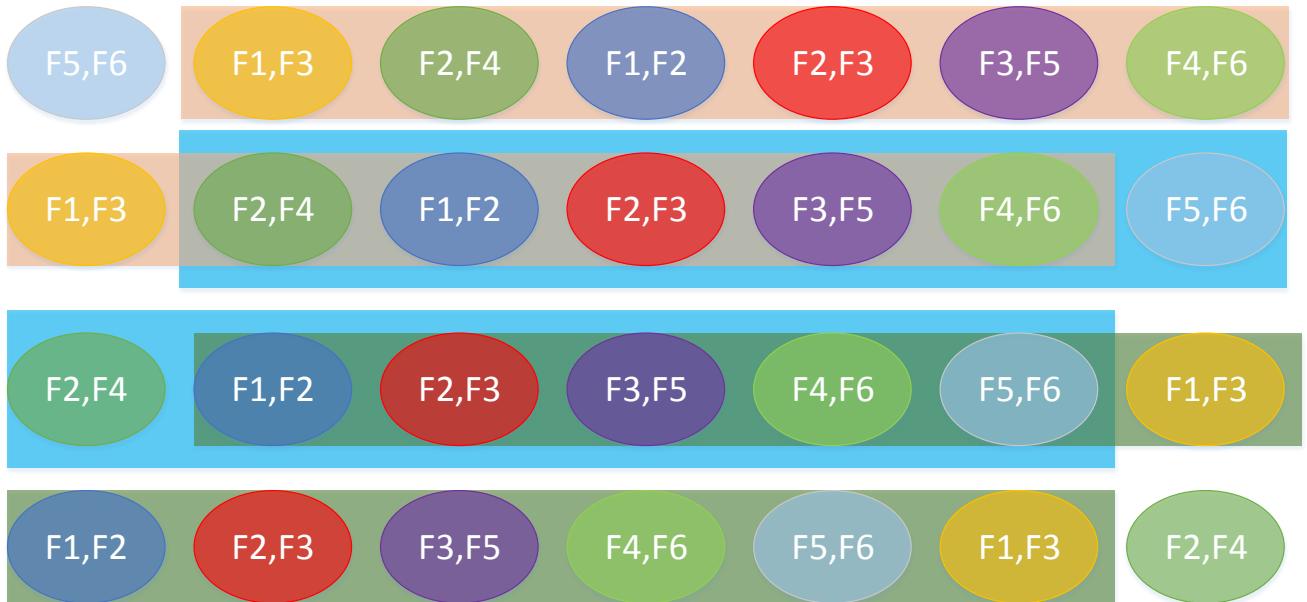


Figure 2: O algoritmo 1 escolhe um vértice(assinalado na Figura 1) e percorre o grafo até o vértice anterior ao inicial. Assim cada vértice encontrado no caminho é colocado na solução e tem sua cobertura de focos somada a cobertura da solução até o momento. Depois o algoritmo pega o próximo vértice no sentido horário e executa novamente. Cada linha da imagem é uma execução iniciando de um vértice (o primeiro vértice da linha). É possível observar através dos retângulos que destacam alguns vértices, que existe sobreposição de problemas. A maior parte do calculo de cobertura de focos já foi calculado na interação anterior.

Algoritmo Gulosso

A chave principal para gerar uma solução gulosa próxima da solução ótima é qual vértice escolher primeiro, pois a partir dai a obrigatoriedade da solução ser conexa induz a escolher os vértices adjacentes. Desta maneira, se o vértice inicial escolhido não fizer parte da solução ótima a solução gulosa não levará ao ótimo uma vez que no melhor dos cenários estará adicionando o vértice inicial à solução ótima caso essa solução ótima seja vizinha do vértice inicial. Sendo assim, foram desenvolvidas duas estratégias gulosas: uma heurística focada na escolha do vértice inicial e uma heurística para a escolha de quais vértices adjacentes serão adicionados à solução gerada pelo algoritmo guloso. A Figura 3(a) mostra uma análise na sobreposição de cobertura de focos na instância dada na especificação. A sobreposição de cobertura é o calculo de quão sobrepostos estão os focos cobertos pelo vértice e seus dois vizinhos, da esquerda e da direita. A Figura 3(b) mostra os possíveis valores de sobreposição para uma distância de tamanho 1 (vizinhos da esquerda e direita imediata).

A heurística gulosa verifica os valores de sobreposição dos vértices e escolhe o vértice com menor valor de sobreposição. Caso haja empate do menor valor, a distância do calculo de sobreposição é incrementada e a análise de sobreposição é feita novamente, porém apenas para os vértices empatados. A análise tem sua distância incrementada até que haja desempate ou até que a distância seja $|V| - 1$ (onde seria calculada a sobreposição para todos os vértices). Se não for possível desempatar a escolha é feita aleatoriamente entre os vértices empatrados.

Após escolher o vértice inicial, a heurística coloca o vértice inicial na solução e começa a escolher quais vértices vizinhos serão parte da solução. A heurística caminha para a direita e para a esquerda verificando a contribuição desse vértice quanto à solução. Em cada interação a heurística compara um par de vértices que são os vértices diretamente vizinhos (à direita e à esquerda) aos vértices que já estão na solução. Se o par de vértices contribui igualmente para a solução, os dois vértices são incluídos na solução. Se um vértice contribui mais, então somente ele é adicionado à solução. A heurística pára de adicionar vértices quando os vértices presentes na solução cobrem todos os focos ou quando já inseriu todos os vértices. O Algoritmo 3

implementa a solução gulosa descrita.

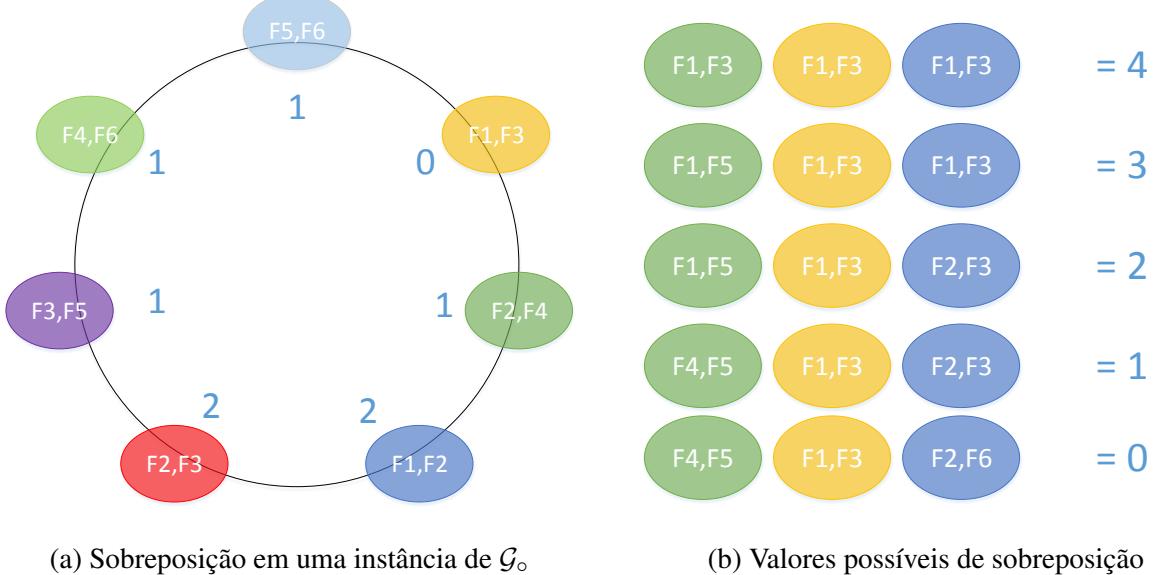


Figure 3: Sobreposição de cobertura de focos com os vizinhos a uma distância de tamanho 1.

Exercício 2

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 **Força-bruta**(1) é dominado pelos loops aninhados das linhas 2 e 4 e pela verificação na linha 5, onde são percorridos todos os r focos para verificar sua cobertura pelo subconjunto atual. O custo de verificar a cobertura portanto é de $O(r)$. No **melhor caso** sua complexidade é $O(r)$ quando o grafo $G(\mathbb{V}, \mathbb{A})$ possui apenas um vértice e portanto só existe um subconjunto \mathbb{V}' e o teste da linha 5 só será executado uma vez. No **pior caso** são gerados todos os subconjuntos de $G(\mathbb{V}, \mathbb{A})$ tendo complexidade de $O((|\mathbb{V}|((|\mathbb{V}| - 1) \cdot r))$.

O tempo de execução do Algoritmo **Programação Dinâmica**(2) é dominado pelos loops aninhados das linhas 2 e 4, que tem complexidade $O(|\mathbb{V}|^2)$. A cada interação do loop interno são executadas duas operações sobre conjuntos. Uma união ou diferença nas linhas 5 a 11 e uma interseção na linha 12 para verificar a cobertura dos focos. A complexidade da união é $O(2 \cdot (|M[i][j - 1]| + |\mathbb{V}|) - 1)$, a complexidade da diferença é $O(2 \cdot (|M[i - 1][j]| + 2) - 1)$ e a complexidade da interseção é $O(2 \cdot (|M[i][j]| + r) - 1)$. Dessa maneira a complexidade das linhas 5 a 13 é dominada pela interseção, pois no pior caso cada voluntário cobre dois focos diferentes e $r = 2|\mathbb{V}|$. A complexidade total é portanto $O(|\mathbb{V}|^2 \cdot r)$.

A análise do Algoritmo **Guloso**(3) será dividida por funções. A função *CalculaSobreposicao* tem complexidade $O(|\mathbb{V}|)$. A função *SomaSobreposicao* tem complexidade $O(1)$. A função *Contribuicao* tem a complexidade dominada pela interseção de conjuntos que nesse caso tem complexidade $O(2 \cdot (|Solucao| + |\mathbb{V}|) - 1)$. Por fim, a função *Guloso* tem complexidade dominada pelos loops da linha 23 à 30. O loop externo tem complexidade no pior caso de $O(|\mathbb{V}|)$. Os loops internos (linhas 24 e 27) tem cada um complexidade no pior caso $O(|\mathbb{V}|)$ onde todos os voluntários tem a mesma sobreposição de cobertura de focos e todos são selecionados como potenciais vértices iniciais da solução gulosa. Assim, a função *Guloso* tem complexidade $O(|\mathbb{V}|^2)$. Portanto conclui-se que o Algoritmo **Guloso**(3) tem complexidade temporal dominada assintoticamente pela função *Guloso* e portanto tem complexidade $O(|\mathbb{V}|^2)$.

Algorithm 3 Guloso

Require: Grafo $\mathcal{G}_o(\mathbb{V}, \mathbb{A})$, Vetor C com a relação $\mathcal{R}(v) : \mathbb{V} \rightarrow F$, Vetor S de sobreposições, Vetor $Solucao$

```
1: function CALCULASOBREPOSICAO( $\mathcal{G}_o, S, C$ )
2:   for all Vértice  $i \in \mathbb{V}$  do
3:     if Foco  $C(i)_1 \subset C(i - 1)$  then
4:        $S(i)++;$ 
5:     if Foco  $C(i)_1 \subset C(i + 1)$  then
6:        $S(i)++;$ 
7:     if Foco  $C(i)_2 \subset C(i - 1)$  then
8:        $S(i)++;$ 
9:     if Foco  $C(i)_2 \subset C(i + 1)$  then
10:       $S(i)++;$ 
11: function SOMASOBREPOSICAO( $SE, S, e, dist$ )
12:    $SE(e)+ = S(e + k) + S(e - k)$ 
13: function CONTRIBUICAO( $C, v, Solucao$ )
14:    $Contribuicao = Solucao \cup C(v)$ 
15:   return  $|Contribuicao| - |Solucao|$ 
16: function GULOSO( $\mathcal{G}_o, S, C$ )
17:   CalculaSobreposicao( $\mathcal{G}_o, S, C, E$ )
18:    $min = \min(S)$ 
19:   for all Vértice  $i \in \mathbb{V}$  do
20:     if  $S(i) == min$  then
21:        $E \leftarrow S(i)$ 
22:    $dist = 1$ 
23:   while  $|escolhidos| > 1$  and  $dist \leq |\mathbb{V}|$  do
24:     for all Vértice  $e \in E$  do
25:       SomaSobreposicao( $SE, S, e, dist$ )
26:        $min = \min(SE)$ 
27:       for all Vértice  $e \in E$  do
28:         if  $SE(e)! = min$  then
29:           retira  $e$  de  $E$ 
30:          $dist++;$ 
31:       if ( $|E| > 1$ )
32:         escolha um  $e \in E$  aleatório
33:          $inicial = e$ 
34:       else
35:          $inicial = E[0]$ 
36:        $Solucao \leftarrow inicial$ 
37:        $apontEsq = inicial - 1$ 
38:        $apontDir = inicial + 1$ 
39:       while  $|Solucao| < |\mathbb{V}|$  &&  $Solucao$  não cobre todos focos do
40:         if  $Contribuicao(C, apontEsq, Solucao) == Contribuicao(C, apontDir, Solucao)$  then
41:            $Solucao \leftarrow apontEsq$ 
42:            $Solucao \leftarrow apontDir$ 
43:            $apontEsq ++$ 
44:            $apontDir ++$ 
45:         else if  $Contribuicao(C, apontEsq, Solucao) > Contribuicao(C, apontDir, Solucao)$  then
46:            $Solucao \leftarrow apontEsq$ 
47:            $apontEsq ++$ 
48:         else
49:            $Solucao \leftarrow apontDir$ 
50:            $apontDir ++$ 
51: return  $Solucao$ 
```

Complexidade Espacial

O **melhor caso** do Algoritmo **Força Bruta** acontece quando existe somente um vértice em $G_o(\mathbb{V}, \mathbb{A})$ e somente um subconjunto é armazenado em $subSet$ e sua complexidade de espaço é $O(1)$. No **pior caso**, todos os subconjuntos são armazenados levando a uma complexidade de espaço de $O(|\mathbb{V}|^2 \cdot |\text{SubSet}|)$. Além disso para armazenar a relação $\mathcal{R}(v) : \mathbb{V} \rightarrow F$ é criado uma estrutura com complexidade $O(2|\mathbb{V}|)$.

O Algoritmo **Programação Dinâmica** tem a complexidade de espaço $O(|\mathbb{V}|^2 \cdot r)$ uma vez que cria uma matriz $|\mathbb{V}| \times |\mathbb{V}|$ com um $SubSet$ composto dos focos cobertos até o momento em cada posição, esse $SubSet$ no **pior caso** tem complexidade r pois todos os focos estão cobertos. Além disso para armazenar a relação $\mathcal{R}(v) : \mathbb{V} \rightarrow F$ é criado uma estrutura com complexidade $O(2|\mathbb{V}|)$.

O Algoritmo **Guloso** tem complexidade de espaço de $O(|\mathbb{V}|^2)$, para armazenar os vértices em uma lista de adjacência, $O(|\mathbb{V}|)$ (pior caso) para armazenar os vértices com menor sobreposição de cobertura de focos e $O(2|\mathbb{V}|)$ para armazenar a relação $\mathcal{R}(v) : \mathbb{V} \rightarrow F$.

Exercício 4

Os algoritmos propostos foram implementados na linguagem C++ e encontram-se no zip juntamente com essa documentação. Foi feito um arquivo .cpp para cada paradigma que pode ser compilado e executado conforme as instruções da especificação do trabalho.

References

- [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).

UFMG – PPGCC/DCC – Projeto e Análise de Algoritmos (PAA)
Primeiro Semestre de 2016 – Paradigmas de Projeto de Algoritmos
Trabalho Prático: ZikaZeroAnelDual [8 pontos]

Nome: Marcus Rodrigues de Araújo

Matrícula: 2016672310

Exercício 1 [3/8 pontos]. Proponha algoritmos de Complexidade Assintótica Polinomial com o Tamanho da entrada (n , m e r) para resolver o Problema ZikaZeroAnelDual usando os seguintes paradigmas:

1. Busca por Força-Bruta

“Tentar todas as combinações de vértices conexos”

```
1  para i = 1 até i <= n faça
2    | janela = new Janela().tamanho(i)
3    | janela.adiciona(1, i-1)
4    | faça
5    |   | se janela é solução então soluções.adiciona(janela) então
6    |   |   | janela.move(sentido_horário)
7    |   enquanto (! janela.percorreu_todo_o_grafo?())
8    |   se soluções.quantidade() > 0 então
9    |   | retorna soluções.melhor_solução()
10   | fim_se
11   fim_para
```

Uma “janela”, no caso deste algoritmo, é uma lista com uma sequência de voluntários. O algoritmo funciona selecionando diferentes tamanhos de janelas e verificando se, para cada tamanho de janela, existe alguma sequência de voluntários que produza um solução para o problema. Se determinada janela limita uma solução, esta é colocada em uma lista de soluções, caso contrário, a janela se “desloca” para o lado a fim de buscar por uma nova possível solução. Se uma janela de tamanho k não consegue limitar nenhuma solução no grafo o tamanho da janela é incrementada e a nova janela percorre todo o grafo novamente.

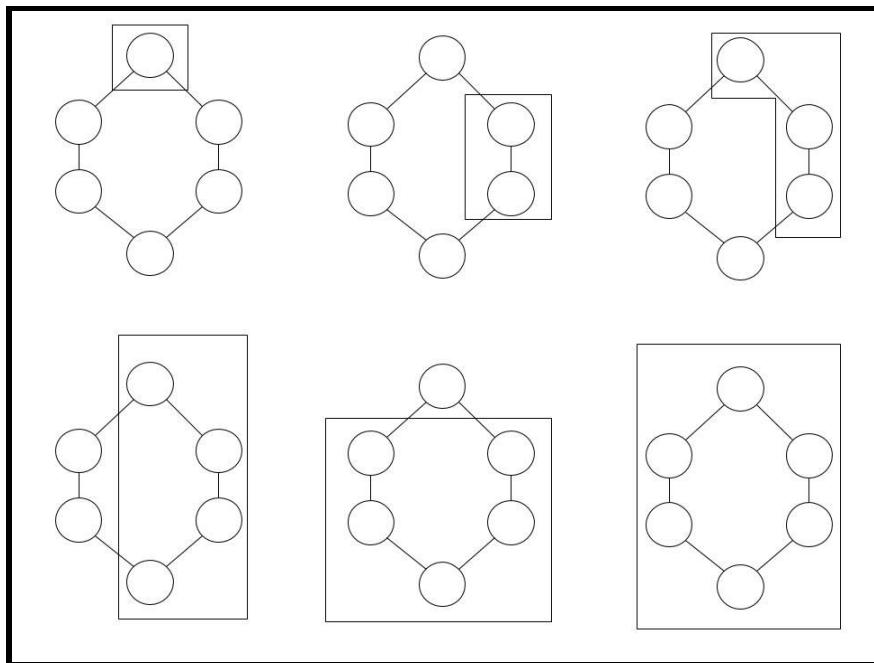


Figura 1. Exemplos de janelas de diferentes tamanhos

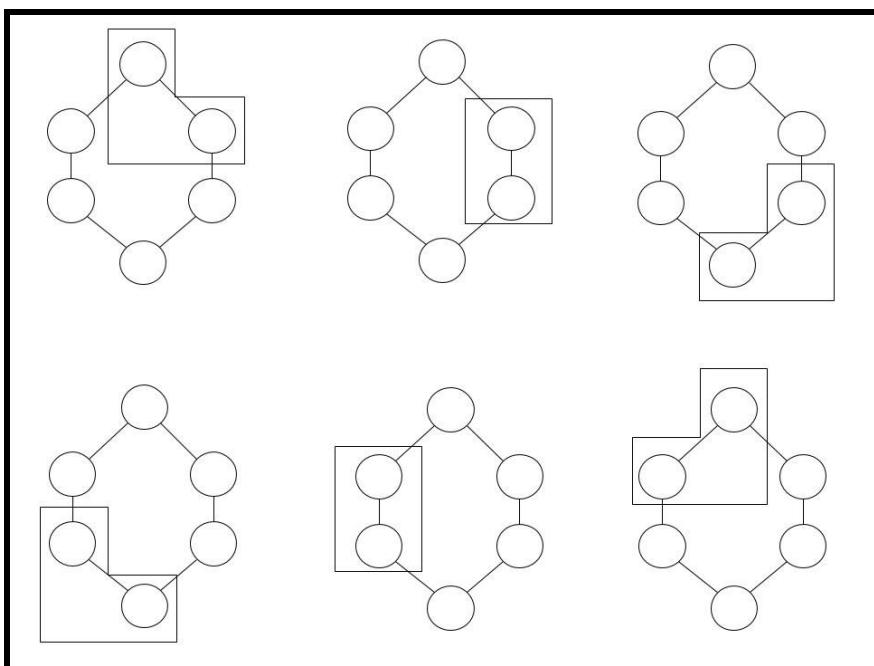


Figura 2. Exemplo de janela de tamanho 2 se “deslocando” pelo grafo.

2. Programação Dinâmica

$solução[i, 0] = \{ v_i \}$

$solução[i, j] = expande(solução[i, j-1]) \text{ se } solução[i, j] \text{ não é solução}$

```
1  para i = 1 até i <= n faça
2    | solução[i] = {V[i]}
3    | solução[i].foi_encontrada = falso
4  fim_para
5  numero_de_soluções = 0
6  enquanto (numero_de_soluções < n) faça
7    | para i = 1 até i <= n faça
8    |   | se (! solução[i].foi_encontrada) então
9    |   |   | se (solução[i].é_solução?()) então
10   |   |   |   | numero_de_soluções = numero_de_soluções + 1
11   |   |   |   | solução[i].foi_encontrada = verdadeiro
12   |   |   | senão
13   |   |   |   | expande(solução[i])
14   |   |   |   | fim_se
15   |   |   | fim_se
16   |   | fim_para
17 fim_enquanto
18 para i = 1 até i <= n faça
19   | solução[i].melhorar_solução()
20   | soluções.adiciona(solução[i])
21 fim_para
22 retorna soluções.melhor_solução()
```

A estratégia dinâmica utilizada considera inicialmente cada nó como uma possível solução. A cada iteração ela verifica se determinado subconjunto forma uma solução. Se o conjunto não é uma solução o algoritmo expande as fronteiras da solução, ou seja, adiciona um vértice a esquerda e outro a direita e volta ao passo de verificação. O algoritmo para de aumentar um padrão quando a solução é encontrada. No fim do algoritmo, quando as soluções a partir de cada nó tiverem sido encontradas, há uma etapa para se tentar “lapidar” cada uma das soluções, removendo possíveis nós em “excesso”. Em seguida, do conjunto de soluções encontradas é selecionada a melhor para ser retornada.

3. Algoritmo Guloso

$solução[i, 0] = \{ v_i \}$

$solução[i, j] = expande_com_melhor_vizinho(solução[i, j-1])$ se $solução[i, j]$ não é solução

```
1 para cada vértice v do grafo faça
2 | solução = {v}
3 | enquanto (! solução.é_solução?()) faça
4 | | expande_de_forma_gulosa(solução)
5 | fim_enquanto
6 | solução.melhorar_solução()
7 | soluções.adiciona(solução)
8 fim_para
9 retorna soluções.melhor_solução()
```

Para cada um dos voluntários do grafo é executada a estratégia gulosa a seguir. Dado um nó específico, primeiramente verifica-se se este nó é solução ou não. Cada vez que um determinado conjunto de nós não forma uma solução, este é expandido de forma a se escolher o voluntário vizinho ao conjunto (a esquerda ou a direita) que cubra mais focos ainda não cobertos. Em caso de empate, se os vizinhos ajudam de igual maneira, selecionam-se os dois. Cada conjunto de vértices gerado é novamente testado para se verificar se eles formam uma solução ou não. Quando uma solução é encontrada, esta é submetida há uma etapa de “lapidação”, como feito na estratégia dinâmica, e incluída ao conjunto de soluções. Depois de se achar uma solução para cada um dos nós do grafo, é selecionado dentre estas a melhor para ser retornada.

Exercício 2 [1/8 pontos]. Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) dos algoritmos propostos no Exercício 1.

1. Busca por Força-Bruta

Temporal

1	para i = 1 até i <=n faça	O(1)	1
2	janela = new Janela().tamanho(i)	O(1)	2
3	janela.adiciona(1, i-1)	O(1)	3
4	faça	-	4
5	se janela.é_solução?() então soluções.adiciona(janela)	O(n)	5
6	n janela.move(sentido_horário)	O(1)	6
7	enquanto (! janela.percorreu_todo_o_grafo?())	O(1)	7
8	se soluções.quantidade() > 0 então	O(1)	8
9	retorna soluções.melhor_solução()	O($n^2 \lg n$)	9
10	fim_se	-	10
11	fim_para	-	11
Total		O(n^3)	

Espacial

1	janela (lista de voluntários)	O(n)	1
2	solução (lista de voluntários)	O(n)	2
3	soluções (lista de soluções)	O(n^2)	3
Total		O(n^2)	

2. Programação Dinâmica

Temporal

1	para i = 1 até i <=n faça	O(1)	1
2	solução[i] = {V[i]}	O(1)	2
3	solução[i].foi_encontrada = falso	O(1)	3
4	fim_para	-	4
5	numero_de_soluções = 0	O(1)	5
6	enquanto (numero_de_soluções < n) faça	O(1)	6
7	para i = 1 até i <=n faça	O(1)	7
8	se (! solução[i].foi_encontrada) então	O(1)	8
9	se (solução[i].é_solução?()) então	O(n)	9
10	numero_de_soluções = numero_de_soluções + 1	O(1)	10
11	solução[i].foi_encontrada = verdadeiro	O(1)	11
12	senão	-	12
13	expande(solução[i])	O(1)	13
14	fim_se	-	14
15	fim_se	-	15
16	fim_para	-	16
17	fim_enquanto	-	17
18	para i = 1 até i <=n faça	O(1)	18
19	solução[i].melhorar_solução()	O(n)	19
20	soluções.adiciona(solução[i])	O(1)	20
21	fim_para	-	21
22	retorna soluções.melhor_solução()	O($n^2 \lg n$)	22
Total		O(n^3)	

Espacial

1	solução[i] (lista de voluntários)	O(n)	1
2	solução[] (lista de lista voluntários)	O(n^2)	2
3	soluções (lista de soluções)	O(n^2)	3
Total		O(n^2)	

3. Algoritmo Guloso

Temporal

1	para cada vértice v do grafo faça	O(1)	1
2	solução = {v}	O(1)	2
3	enquanto (! solução.é_solução?()) faça	O(n)	3
4	n expande_de_forma_gulosa(solução)	O(1)	4
5	fim_enquanto	-	5
6	solução.melhorar_solução()	O(n)	6
7	soluções.adiciona(solução)	O(1)	7
8	fim_para	-	8
9	retorna soluções.melhor_solução()	O($n^2 \lg n$)	9
		Total	O(n^3)

Espacial

1	solução (lista de voluntários)	O(n)	
2	soluções (lista de soluções)	O(n^2)	
		Total	O(n^2)

Exercício 3 [3/8 pontos]. Implemente o algoritmos propostos no Exercício 1 na linguagem de programação C, C++, Java ou Python.

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

Exercício 4 [1/8 ponto]. Execute testes da implementação do Exercício 3, propondo instâncias interessantes para o Problema ZikaZeroAnelDual. Uma sugestão é que seja produzido um gráfico do tempo de execução por Tamanho da entrada (n , m e r) comparando os três algoritmos implementados.

Para validar os algoritmos e as análises mostradas anteriormente foram feitos diferentes tipos de testes. Para auxiliar nesse processo foi construído um gerador de casos de teste que atribui os focos aos usuários de forma aleatória. O gerador recebe como entrada o número de voluntários e o número de focos considerados.

Os testes foram divididos em dois grupos principais. O primeiro deles é composto por casos de teste em que temos o mesmo número de voluntários e de focos. Diferentemente do primeiro grupo, o segundo é composto por testes onde se tem mais voluntários do que focos.

Abaixo, seguem os dados dos casos testes realizados e dos tempos de execução de cada um dos programas:

Nro Voluntários	Nro Focos	Tamanho da Resposta	Tempo B	Tempo D	Tempo G	Média
10	10	8	0.004	0.004	0.004	0.004
20	20	16	0.003	0.003	0.003	0.003
30	30	27	0.005	0.005	0.006	0.0053
40	40	33	0.013	0.009	0.012	0.0113
50	50	42	0.022	0.014	0.007	0.0143
60	60	53	0.033	0.024	0.011	0.0227
70	70	63	0.044	0.029	0.017	0.03
80	80	70	0.056	0.04	0.018	0.038

Tabela 1. Configuração dos testes realizados (primeiro grupo)

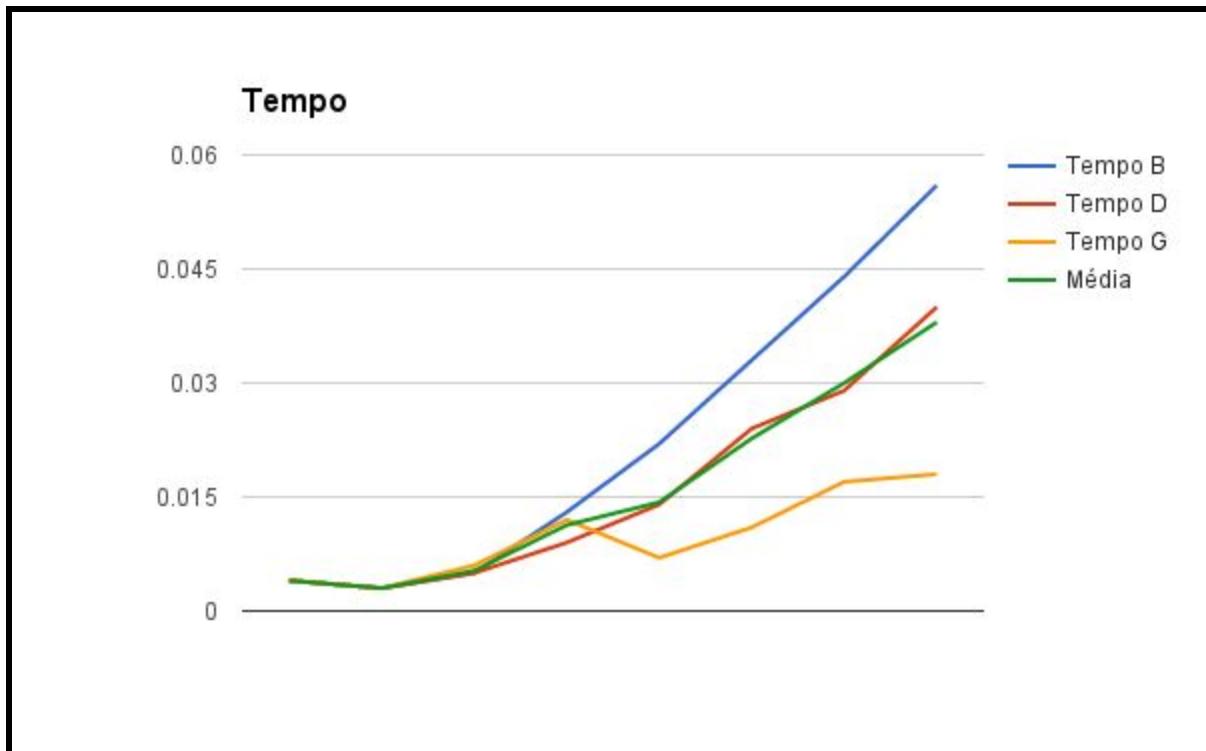


Gráfico 1. Gráfico construído com os tempos de execução da tabela 1

Nro Voluntários	Nro Focos	Tamanho da Resposta	Tempo B	Tempo D	Tempo G	Média
320	160	265	2.063	1.75	0.464	1.4257
420	240	369	5.362	4.396	0.682	3.48
500	250	374	6.497	6.14	1.315	4.6507
640	320	554	17.28	14.63	2.128	11.346
850	420	761	42.272	32.952	3.975	26.3997
1000	500	919	71.786	50.868	5.717	42.7903
1280	640	1200	155.035	117.143	9.508	93.8953

Tabela 2. Configuração dos testes realizados (segundo grupo)

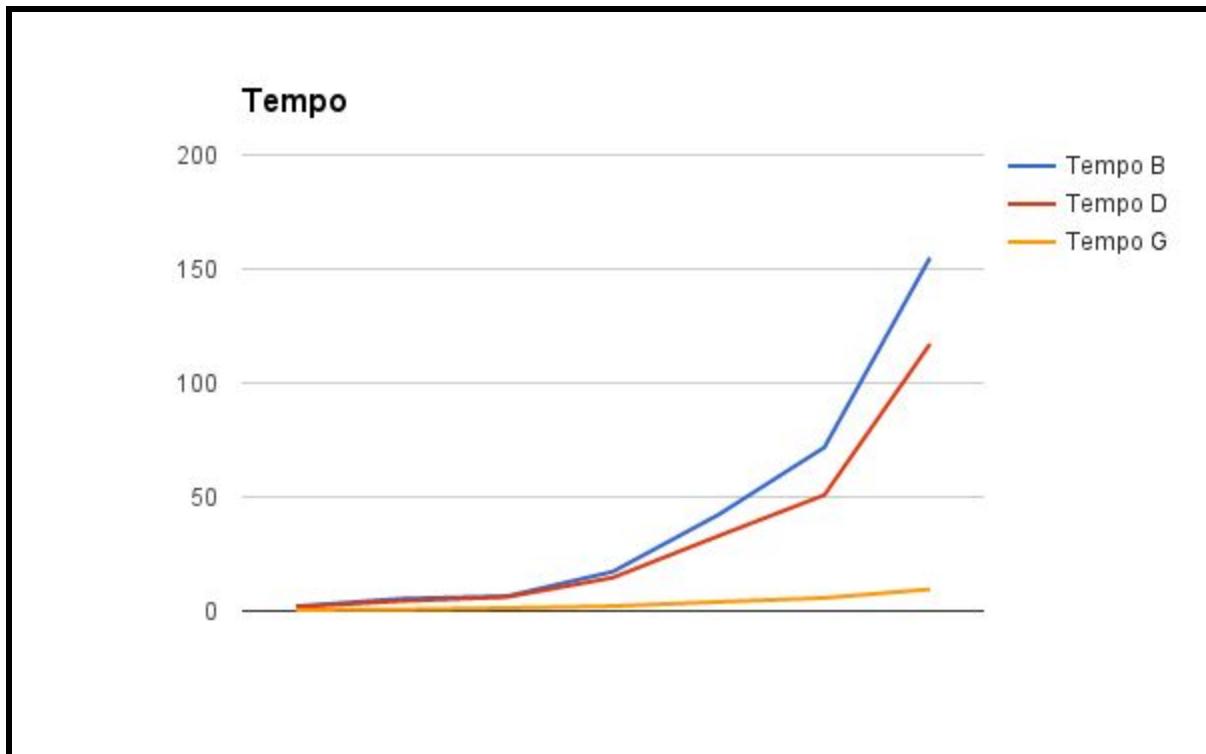


Gráfico 2. Gráfico construído com os tempos de execução da tabela 2

Exercício 5 [1/8 pontos]. Compare a análise e a execução, respectivamente, Exercícios 2 e 4. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

Apesar dos três algoritmos apresentados possuírem a mesma complexidade assintótica, $O(n^3)$, é possível verificar, a partir dos dados apresentados anteriormente, a existência de algoritmos que são melhores do que os outros. O algoritmo guloso apresentou o melhor desempenho geral em relação ao tempo de execução e somente em um dos casos de teste apresentou uma resposta diferente da ótima, que sempre foi encontrada pelos outros dois algoritmos.

Além disso, é possível verificar que a estratégia de programação dinâmica apresentou tempos de execução muito próximos da média dos tempos dos três algoritmos. Em todos os testes realizados, a estratégia de programação dinâmica sempre retornou a resposta ótima e sempre foi mais rápida quando comparada a estratégia por força bruta.

Como a força bruta testa todas as combinações existentes tem-se que essa estratégia sempre encontra a resposta ótima para o problema. Neste trabalho prático, esse fato foi explorado, uma vez que o algoritmo de força bruta pode ser considerado como principal baseline para as outras duas estratégias.

Por fim, os dados mostrados na seção anterior eram esperados mesmo com a complexidade dos algoritmos sendo basicamente a mesma. Para uma análise mais cirúrgica seria necessário a função de complexidade de cada um dos algoritmos.

Trabalho Prático - Paradigmas de Programação

Renato Florentino Garcia
2016661865

21 de junho de 2016

UFMG
DCC865 PG Projeto e Análise de Algoritmos
For: Prof. Wagner Meira

1 Definições

Para os problemas que se seguem, serão adotadas as seguintes definições: $G(V, A)$ é um grafo em anel, com um conjunto de vértices $V = \{v_1, v_2, \dots, v_n\}$ representando n voluntários, e um conjunto de arestas $A = \{a_1, a_2, \dots, a_m\}$ representando os m laços de amizade. Devido ao fato de G ser um grafo em anel segue que $m = n = |A| = |V|$.

Sem perda de generalidade, será assumido que cada vértice v_i estará conectado aos vértices v_{i-1} e v_{i+1} . Excepcionalmente para se permitir o ciclo, o vértice v_1 estará conectado aos vértices v_n e v_2 e o vértice v_n aos vértices v_{n-1} e v_1 .

Juntamento ao grafo G , existe um conjunto $F = \{f_1, f_2, \dots, f_r\}$ com os r focos de reprodução dos mosquitos, e uma função $R : V \rightarrow F^2$ que mapeia cada um dos vértices para os dois focos que são acessados por cada um dos voluntário.

2 Exercício 1

Dois dois métodos listados abaixo utilização como função os algoritmos listado a seguir:

Data: $G(V, A)$
Result: U , conjuntos com todas as combinações de voluntários passíveis de serem uma solução.
1 **foreach** $i \in \{1, 2, \dots, m\}$ **do**
2 **foreach** $w \in \{0, 1, 2, \dots, m\}$ **do**
3 $| U \leftarrow U \cup \{v_i, v_{(i+1)\%m}, \dots, v_{(i+w)\%m}\}$
4 **end**
5 **end**
6 **return** U

Algoritmo 1: Conjunto universo

2.1 Força Bruta

Data: $G(V, A), F, R(v)$
Result: V_z subconjunto de voluntários que formam a solução do problema
1 **foreach** $V_s \in U_g$, onde U_g é o conjunto universo do grafo $G(V, A)$ **do**
2 $| F_s \leftarrow \bigcup_{v \in V_s} R(v)$
3 **if** $F_s = F$ **then**
4 $| \text{return } V_s$
5 **end**
6 **end**

Algoritmo 2: Solução por força bruta

Note que a linha 2 constrói um conjunto $F_s \subseteq F$ que contém todos os focos acessados por todos os vértices presentes no subconjunto $V_s \subseteq V$.

2.2 Programação Dinâmica

Data: $G(V, A), F, R(v)$
Result: V_z subconjunto de voluntários que formam a solução do problema

```

1  $S_u \leftarrow \{\}$ 
2 foreach  $V_s \in U_g$ , onde  $U_g$  é o conjunto universo do grafo  $G(V, A)$  do
3   |  $S_u \leftarrow S_u \cup \{\bigcup_{v \in V_s} R(v)\}$ 
4 end
5  $P_f \leftarrow \mathbb{P}(F)$ , o conjunto potência dos focos
6  $T \leftarrow$  matriz com dimensões  $m + 1 \times |P_f|$ , e elementos  $(\infty, \emptyset)$ 
7  $T[0][\emptyset] \leftarrow 0$ 
8 foreach  $i \in 1, 2, \dots, m$  do
9   | foreach  $S \in S_u$  do
10    |   |  $A \leftarrow T[i - 1][S \setminus R(v_i)]$ 
11    |   |  $B \leftarrow T[i - 1][S]$ 
12    |   | if  $A[0] + 1 < B[0]$  then
13    |   |   |  $T[i][S] \leftarrow (A[0] + 1, A[1] \cup v_i)$ 
14    |   | end
15    |   | else
16    |   |   |  $T[i][S] \leftarrow B$ 
17    |   | end
18   | end
19 end
20 return  $T[m][F][1]$ 
```

Algoritmo 3: Solução por programação dinâmica

2.3 Gulosso

Data: $G(V, A), F, R(v)$
Result: V_z subconjunto de voluntários que formam a solução do problema

```

1  $v_m \leftarrow v$  escolhido aleatoriamente de  $V$ 
2  $V_s \leftarrow \{v_m\}$ 
3  $F_s \leftarrow R(v_m)$ 
4  $v_l, v_r \leftarrow$  os dois vértices conexos à  $v_m$ 
5 while  $F_s \neq F$  do
6   |  $F_l \leftarrow F_s \cup R(v_l)$ 
7   |  $F_r \leftarrow F_s \cup R(v_r)$ 
8   | if  $|F_l| > |F_r|$  then
9   |   |  $V_s \leftarrow V_s \cup \{v_l\}$ 
10  |   |  $F_s \leftarrow F_l$ 
11  |   |  $v_l \leftarrow v : v \in \{\text{vértices conexos à } v_l\} \text{ e } v \notin V_s$ 
12  | end
13  | Else  $V_s \leftarrow V_s \cup \{v_r\}$ 
14  |  $F_s \leftarrow F_r$ 
15  |  $v_r \leftarrow v : v \in \{\text{vértices conexos à } v_r\} \text{ e } v \notin V_s$ 
16 end
17 return  $V_s$ 
```

Algoritmo 4: Solução usando algoritmo gulosso

3 Exercício 2

3.1 Conjunto universo

O algoritmo 1, conjunto universo, possui uma complexidade temporal $O(V^2)$, por causa dos dois laços nas linhas 1 e 2. A complexidade espacial é $O(V^3)$, pois a cada iteração o conjunto U é aumentado com $|V|$ elementos do tamanho da janela.

3.2 Força Bruta

O algoritmo 2, executa em $O(V^2)$, pois itera em cada elemento do conjunto universo. A complexidade espacial é a mesma do algoritmo 1, $O(V^3)$

3.3 Programação Dinâmica

O algoritmo de Programação dinâmica (algoritmo 3) tem tempo de execução $O(V^3)$, pois o laço da linha 9 é executado $|V|$ vezes, e ele próprio é $O(V^2)$. A maior estrutura é uma matriz com $O(V)$ linhas e $O(V^2)$ colunas (caso se implemente como uma matriz esparsa que acrescente colunas sob demanda), e esta matriz define complexidade espacial do algoritmo como $O(V^3)$.

3.4 Gulosso

O algoritmo gulosso tem complexidade temporal como $O(V)$, pois itera uma única vez sobre os vértices, e complexidade espacial $O(V)$, pois a maior estrutura que armazena a parte do grafo são os vértices que compõem a solução.

4 Exercício 3

Os códigos implementados em Python estão anexos.

Trabalho Prático de Paradigmas - PAA

Leandro T. C. Melo

¹DCC - UFMG

ltcmelo@dcc.ufmg.br

Resumo. Este é o relatório do Trabalho Prático de Paradigmas, da disciplina PAA. Ele consiste da resolução do Problema ZikaZeroAnelDual e exercícios relacionados.

1. Exercício 1

Algoritmos de complexidade assintótica polinomial.

• **Força-Bruta**

O algoritmo proposto define inicialmente uma orientação de travessia do grafo anel: horário ou anti-horário. Em seguida o algoritmo seleciona, a cada iteração, um vértice do grafo anel e, a partir dele e através orientação definida, percorre o grafo vizinho-a-vizinho coletando seus focos correspondentes até que todos os focos sejam encontrados (ou, se não houver solução por que existe mais focos do aqueles de fato associados a vértice do grafo, quando todos os vértices tenham sido visitados). Note que é irrelevante, do ponto de vista do resultado, se é a orientação horário ou anti-horário que é utilizada, o importante é que tal orientação seja consistente por todos as travessias. Caso contrário, pode-se não passar pela solução ótima no espaço de soluções. Com uma análise agregada pode-se calcular a complexidade de tempo de $O(V^2 \lg V)$ para este algoritmo. Sua composição básica é de dois loops: um deles percorrerá sempre todos os V vértices do grafo enquanto que o outro percorrerá todos os outros $V - 1$ vértices do grafo, no pior caso (idealmente todos os focos serão encontrados antes que os $V - 1$ vértices restantes sejam visitados). O loop interno conta com um passo de ordenação dos sequências de vértices encontrados (os candidatos), o que contribuiu com o fator $O(\lg V)$ - os focos podem ser contabilizados em uma tabela de hash, com fator de inserção/verificação constante amortizado. Ao final do processo, tendo sido todas as possíveis sequências de candidatos agrupadas em um contêiner, ainda é necessário uma ordenação para escolher a menor. Mas de qualquer maneira, esse fator $O(V \lg V)$ já é absorvido pelo fator $O(V^2 \lg V)$. A complexidade de espaço do algoritmo também é $O(V^2)$. Basicamente, a única informação armazenada pelo algoritmo em si são as possíveis sequências "candidatas" de vértices, as quais são comparadas posteriormente afim de se escolher a melhor de acordo com os critérios no enunciado do TP - o algoritmo propriamente dito não armazena os focos, apenas os utiliza para detectar se determinado par de focos é ainda desconhecido. Como há o máximo de V vértice em cada sequência candidata e há no máximo V sequências candidatas, então o espaço ocupado é $O(V^2)$. O pseudocódigo é mostrado pelo algoritmo 1.

Algorithm 1 ForcaBruta

Require: Grafo anel $G(V,E)$ e conjunto F de r focos.

Ensure: Menor conjunto V' tal que todo r é acessado e o grafo induzido por V' em G é conexo.

- 1: Defina orientação O para coleta de focos adjacentes.
 - 2: Cria um container C para todas as sequências candidatas.
 - 3: **for** $v \in V$ **do**
 - 4: Crie uma tabela de hash $Found$ para os focos encontrados nesta iteração.
 - 5: Crie uma sequência candidata S vazia.
 - 6: **while** $Found \neq AllFocuses$ AND $S \neq V$ **do**
 - 7: $S \leftarrow S \cup v$
 - 8: Obtenha os focos $\{r1, r2\}$ de v .
 - 9: $Found \leftarrow Found \cup \{r1, r2\}$ // A tabela desconsidera duplicatas.
 - 10: $v \leftarrow next(v, O)$ // Pega o próximo de acordo com orientação.
 - 11: **end while**
 - 12: **if** $Found \neq AllFocuses$ **then**
 - 13: $C \leftarrow C \cup S$
 - 14: **end if**
 - 15: Ordene os candidatos em C e pegue o melhor (no caso, o primeiro)
 - 16: **end for**
-

- **Programação Dinâmica**

O algoritmo de programação dinâmica proposto possui basicamente o mesmo esqueleto daquele de força-bruta. No entanto, faz-se uso de uma *cache* que armazena uma coleção *FocusCol* de focos já conhecidos para travessias (parciais) já realizadas no grafo. Essa cache permite que, ao selecionar um vértice v para a iteração corrente, o algoritmo pode dar um "pulo" para um vértice w , transitivamente vizinho a v , caso o caminho $\mathcal{C} v \rightsquigarrow u \rightsquigarrow w$ já tenha sido anteriormente percorrido. Neste caso os focos coletados ao longo de \mathcal{C} são pré-populados na tabela *Found* da iteração em questão. Considere, por exemplo, a figura 1 que mostra o grafo do enunciado do TP. Suponha que a orientação de travessia é anti-horário e primeira iteração do algoritmo começa com o vértice $v1$. Todos os focos serão encontrados apenas ao chegar no vértice $v2$, delimitando o caminho $\mathcal{C} = v1 -> v4 -> v6 -> v5 -> v2$. Suponha agora que o vértice inicial da próxima iteração seja o vértice $v4$. Ao invés de percorremos um caminho parcial já conhecido, podemos pular diretamente para o vértice $v2$ e contabilizar quais são os focos coletados ao longo deste caminho. A cache é uma tabela de $V \times V$ vértices onde cada célula armazena os focos já conhecidos ao se percorrer um caminho $v \rightsquigarrow u$. O objetivo é encontrar nesta tabela qual o vértice *alcançável* u mais distante de v que tenha o maior número de focos conhecidos - naturalmente, a tabela é originalmente vazia e as chances de encontrarmos tais caminhos depende da entrada do programa. A complexidade de tempo do algoritmo dinâmico é mesma do de força bruta, pois o pior caso se mantém, ou seja $O(V^2 \lg V)$. As operações de salvar e restaurar os dados da cache contribuem com fator constante amortizado, já que o armazenamento é feito em um array bi-dimensional. No entanto, a complexidade de espaço agora é maior devido ao cache, que armazena $V \times V$ vértices

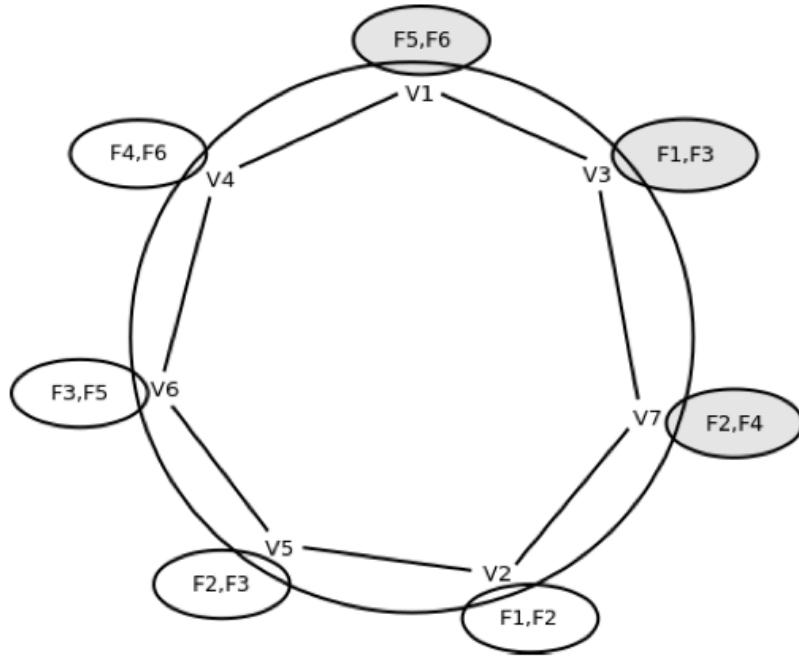


Figure 1. Grafo original do enunciado do TP.

e os focos coletados por caminhos parciais. Portanto, a complexidade de espaço é $O(V^2 + VF)$. O pseudocódigo é mostrado pelo algoritmo 2.

- **Guloso**

O algoritmo guloso proposto seguem o princípio básico dos algoritmos anteriores no sentido de que ele promove travessias para todos os vértices do grafo. No entanto, ele faz uma *escolha gulosa* logo no início da iteração baseada *localmente* nos focos associados aos dois vizinhos do vértice v inicial desta iteração. Ou seja, ao contrário dos outros dois algoritmos onde é necessário definir uma orientação para a travessia, no caso do algoritmo guloso essa orientação é definida por um critério local oriundo da relação de focos associados ao vértice v e seus dois vizinhos: aquele que possuir o maior número de focos diferentes é o escolhido (em caso de empate a escolha é arbitrária). Esse algoritmo só levará sempre a solução ótima para grafos onde exista caminhos de no máximo 3 vértices que contemplam todos os focos: considere um caminho $C = v -> w -> w$; a solução ótima é com certeza encontrada ou a partir do vértice v ou a partir do vértice w ; uma das duas orientações corresponde àquela realizada pelo algoritmo guloso. Caso contrário, a escolha local feita no início da iteração pode ser sub-ótima. A complexidade assindética deste algoritmo é igual ao de força-bruta, $O(V^2 \lg V)$, tanto no espaço como no tempo. No entanto, na prática seu comportamento é mais eficiente pois não requer o cálculo prévio de orientação. O pseudocódigo é mostrado pelo guloso 3. **NOTA:** No dia 15/06 foi postado no fórum que a complexidade do algoritmo guloso deveria ser melhor do que a do força-bruta, visto que, não fosse esse o caso, por quê utilizar o algoritmo guloso. Enquanto que essa afirmação faz sentido, ela não estava presente no enunciado original do TP. Além disso, meu ob-

Algorithm 2 Dinâmico

Require: Grafo anel $G(V, E)$ e conjunto F de r focos.

Ensure: Menor conjunto V' tal que todo $r \in F$ é acessado e o grafo induzido por V' em G é conexo.

- 1: Inicializa *cache* vazia.
- 2: Defina orientação O para coleta de focos adjacentes.
- 3: Cria um container C para todas as sequências candidatas.
- 4: **for** $v \in V$ **do**
- 5: Crie uma tabela de hash $Found$ para os focos encontrados nesta iteração.
- 6: Crie uma sequência candidata S vazia.
- 7: Verifique se $u = cache.load(v)$ permite "pularmos" adiante.
- 8: **if** u é válido **then**
- 9: Pré-popula C e $Found$
- 10: $v \leftarrow u$
- 11: **end if**
- 12: **while** $Found \neq AllFocuses$ AND $S \neq V$ **do**
- 13: $S \leftarrow S \cup v$
- 14: Obtenha os focos $\{r1, r2\}$ de v .
- 15: $Found \leftarrow Found \cup \{r1, r2\}$ // A tabela desconsidera duplicatas.
- 16: Salve na cache $cache.store(S, \{r1, r2\})$
- 17: $v \leftarrow next(v, O)$ // Pega o próximo de acordo com orientação.
- 18: **end while**
- 19: **if** $Found \neq AllFocuses$ **then**
- 20: $C \leftarrow C \cup S$
- 21: **end if**
- 22: Ordene os candidatos em C e pegue o melhor (no caso, o primeiro)
- 23: **end for**

jetivo primordial nas implementações foi em exercitar o aspecto fundamental das diferenças dos paradigmas, o que é refletido nesta proposta do algoritmo guloso e de sua escolha local. Caso tivesse maior disponibilidade de tempo, proporia e implementaria um algoritmo guloso que fizesse uma etapa de pré-processamento no grafo para identificar o maior diferença de conjuntos de focos em um dado caminho de no máximo até X vértices (onde X seria um número definido de maneira ad-hoc). Visto que me programei para a data original de submissão do TP, dia 17/06, infelizmente não poderei fazer tal modificação.

Algorithm 3 Guloso

Require: Grafo anel $G(V,E)$ e conjunto F de r focos.

Ensure: Menor conjunto V' tal que todo r é acessado e o grafo induzido por V' em G é conexo.

- 1: Cria um container C para todas as sequências candidatas.
- 2: **for** $v \in V$ **do**
- 3: Crie uma tabela de hash $Found$ para os focos encontrados nesta iteração.
- 4: Crie uma sequência candidata S vazia.
- 5: Obtenha os focos do vizinho de v à direita, F_{right} , e à esquerda, F_{left} .
- 6: **if** F_{right} contempla mais "novos" focos que F_{left} **then**
- 7: Defina orientação "à direita".
- 8: **else**
- 9: Defina orientação "à esquerda".
- 10: **end if**
- 11: **while** $Found \neq AllFocuses$ AND $S \neq V$ **do**
- 12: $S \leftarrow S \cup v$
- 13: Obtenha os focos $\{r1, r2\}$ de v .
- 14: $Found \leftarrow Found \cup \{r1, r2\}$ // A tabela desconsidera duplicatas.
- 15: $v \leftarrow next(v, O)$ // Pega o próximo de acordo com orientação.
- 16: **end while**
- 17: **if** $Found \neq AllFocuses$ **then**
- 18: $C \leftarrow C \cup S$
- 19: **end if**
- 20: Ordene os candidatos em C e pegue o melhor (no caso, o primeiro)
- 21: **end for**

2. Exercício 2

A análise de complexidade dos algoritmos foi feita em conjunto com suas respectivas descrições no Exercício 1.

3. Exercício 3

A implementação dos 3 algoritmos foi realizada em C++ e todo código-fonte segue em anexo. Sua documentação está embutida como comentários no próprio código-fonte. NOTA: No arquivo algo.h há uma flag para habilitar/desabilitar o *trace* da evolução dos algoritmos. Caso necessário, ela pode ser habilitada para inspeção mais detalhada de seus passos.

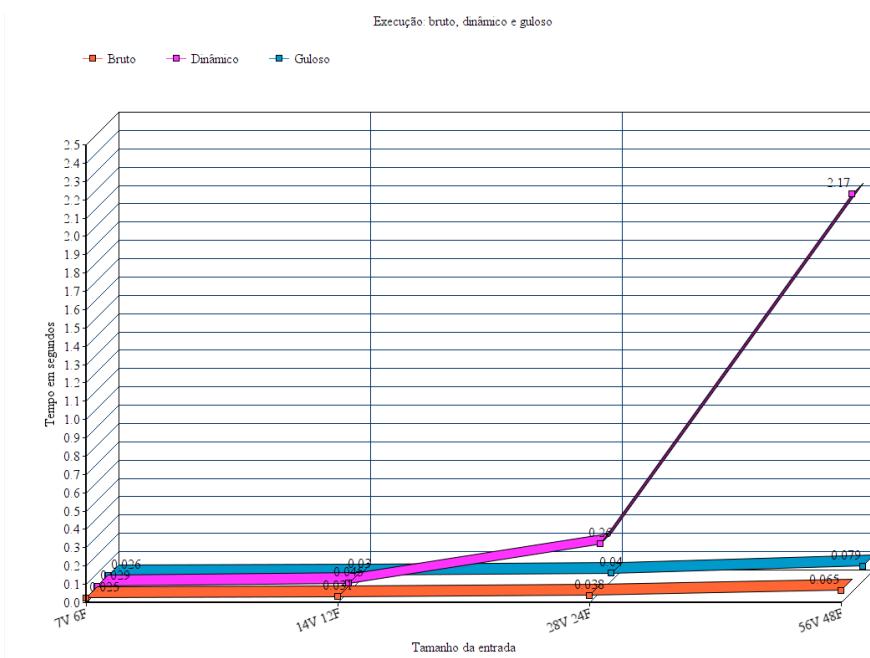


Figure 2. Benchmark das implementações.

4. Exercício 4

Foram realizados testes de execução em que o número de vértices do grafo e o número de focos foram iterativamente dobrados. Visto que os algoritmos possuem complexidade $O(V^2)$, espera-se observar que o tempo de execução se modifique de maneira quadrática. A figura 2 ilustra o comportamento das 3 implementações perante a essas modificações na entrada.

5. Exercício 5

Os tempos de execução encontrados não correspondem à expectativa assintótica. Na verdade, os valores estão relativamente distantes do esperado, pois o tempo de execução não demonstrou o crescimento quadrático esperado. Avaliando a implementação como um todo e seu perfil de execução no sistema, argumento que isso se deve ao fato dos grafos serem ainda bastante pequenos. Ou seja, as operações de acesso a memória e processamento como um todo, "absorvem" a complexidade assintótica. Apenas no caso do algoritmo dinâmico, como o volume de dados aumenta consideravelmente, conseguimos observar uma tendência na figura 2. Mas ainda assim seriam necessários testes com grafos significativamente maiores para que observarmos o crescimento quadrático (e logarítmico) do algoritmo.

Trabalho Prático de Paradigmas de Projetos de Algoritmos

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. Descrição do Problema

O problema do **ZicaZeroAnelDual** é caracterizado por um grafo anel $G(V, A)$, em que V é um conjunto de n voluntários, A é um conjunto de $m = |V|$ laços de amizade, F é um conjunto dos r focos de reprodução do mosquito, $R(v) : V \rightarrow F$ é uma relação definida para cada $v \in V$, explicitando os focos de reprodução aos quais o voluntário v tem acesso, sendo que $|R(v)| = 2, \forall v \in V$. 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 de V' em G seja conexo.

2. Algoritmos

Nesta seção, é apresentado os algoritmos propostos de acordo com cada paradigma solicitado, como visto em [Cormen 2009]. Foram implementados os algoritmos nos seguintes paradigmas:

- Algoritmo de Busca por Força Bruta
- Algoritmo Guloso
- Algoritmo de Programação Dinâmica

2.1. Busca por Força Bruta

O algoritmo de Busca por Força Bruta é baseado na implementação simples de uma busca em largura em um grafo, onde todos os vizinhos de um vértice são visitados.

Usar uma busca em largura em um grafo anel para encontrar a resposta parte de uma premissa que se o vértice do meio da solução for a raiz da busca em largura, a solução ótima será encontrada, portanto é feita a busca em largura à partir de todos os vértices.

O algoritmo *Get_Smaller_Sum* utilizado abaixo verifica em uma lista de respostas ótimas qual delas tem a menor soma de identificadores, com um custo máximo de $O(V)$.

Ao chegar em uma solução, o processo de busca em largura é interrompido e a solução é armazenada no vetor *best*. Se houver outra solução de mesmo tamanho, ela também é colocada vetor *best*, porém se a solução tiver um tamanho menor, as soluções anteriores são descartadas.

Algoritmo 1: Resolução do **ZicaZeroAnelDual** por Força Bruta

```
1 Algoritmo Força_Bruta( $V, R$ ) :
2    $best \leftarrow \{V\}$ 
3   for  $v \in V$  do
4     Faça uma Busca em Largura com raiz em  $v$ , acumulando os focos visitados
      e interrompendo-a ao completar todos os focos;
5     Seja  $S$  o resultado desta Busca em Largura;
6     if  $|S| \leq |best[0]|$  then
7       if  $|S| < |best[0]|$  then
8          $best \leftarrow \emptyset$ 
9        $best.push(S)$ 
10  return Get_Smaller_Sum( $best$ )
```

2.2. Algoritmo Guloso

O algoritmo guloso proposto por este trabalho é uma heurística, ou seja, não tem garantia de resultado ótimo. Entretanto, seu tempo execução é apenas $O(n)$.

A proposta segue a seguinte ideia: Primeiramente, para todas as arestas $(u, v) \in A$, atribuia um peso $w(u, v)$ sendo esse valor $w(u, v) = \frac{1}{|R(u) \cup R(v)|}$. Este peso caracteriza a elegibilidade desta aresta para um subproblema local.

Após isso, ordene o conjunto A de arestas de acordo com seu peso. A aresta (u, v) que conter o menor peso, será considerada como parte da solução e os vértices u e v são adicionados à solução parcial. A partir deste ponto, é verificado se o vizinho de u à esquerda visita mais focos que o vizinho à direita de v . Se sim, ele é adicionado à solução parcial e o processo continua desta maneira.

A heurística encerra o processo quando uma solução válida é alcançada e toma ela como solução ótima. Abaixo, está a descrição do algoritmo guloso proposto:

Algoritmo 2: Resolução do **ZicaZeroAnelDual** por Estratégia Gulosa

```
1 Algoritmo Algoritmo_Guloso( $V, A, R$ ) :
2   Calcule todos os pesos das arestas onde cada  $(u, v)$  terá um peso de  $\frac{1}{|R(u) \cup R(v)|}$ ;
3   Seja  $S$  uma lista vazia e adicione os vértices  $u$  e  $v$  da aresta de menor custo;
4   while  $S$  não for uma solução válida do
5     Seja  $left$  o conjunto  $S$  mais o vértice vizinho à esquerda;
6     Seja  $right$  o conjunto  $S$  mais o vértice vizinho à direita;
7     if  $|\bigcup_{i \in left} R(i)| > |\bigcup_{j \in right} R(j)|$  then
8        $S \leftarrow left$ 
9     else
10       $S \leftarrow right$ 
11  return  $S$ 
```

2.3. Programação Dinâmica

O algoritmo de programação dinâmica proposto para este trabalho funciona da seguinte maneira: Temos uma matriz H , onde as linhas representam a lista conjuntos das partes de tamanho i e cada j é um subgrafo possível de tamanho i . Entretanto, a construção é feita incrementalmente, criando no máximo n^2 subgrafos conexos possíveis em um grafo anel.

A execução termina no momento que um subgrafo gerado visita todos os focos de zica do problema, evitando a criação de subgrafos desnecessários. No algoritmo abaixo, está especificado a construção deste paradigma:

Algoritmo 3: Resolução do ZicaZeroAnelDual por Programação Dinâmica

```

1 Algoritmo Algoritmo_Dinâmico( $V, A, F, R$ ) :
2    $H \leftarrow \{\{\}\};$ 
3   for  $v \in V$  do
4      $\quad H[0].push(\{v\});$ 
5    $best \leftarrow \emptyset;$ 
6   for  $k \leftarrow 0$  to  $|V|$  do
7     for  $i \leftarrow 0$  to  $|V|$  do
8       if  $i + 1 < |V|$  then
9          $\quad j \leftarrow i + 1;$ 
10      else
11         $\quad j \leftarrow 0;$ 
12       $new \leftarrow H[k][i] \cup \{H[k][j][k]\};$ 
13      if  $|\bigcup_{k \in new} R(k)| = |F|$  then
14         $\quad best.push(new);$ 
15      else
16         $\quad H[k + 1].push(new);$ 
17      if  $|best| > 0$  then
18         $\quad break;$ 
19   return Get_Smaller_Sum(best);

```

3. Análise de Complexidade

3.1. Busca por Força Bruta

É sabido que a busca em largura em um grafo possui uma ordem de complexidade $O(|V| + |E|)$. O grafo destre problema possui uma característica específica de ser um grafo anel, logo temos que $|V| = |E|$. Portanto, a complexidade de se fazer uma busca em largura neste tipo de grafo resulta em $O(2|V|)$.

O processo de busca em largura utilizando uma pilha de vértices é apresentado em [Cormen 2009] e precisa ser feito a partir de cada vértice pertencente do grafo em busca de uma solução ótima, fazendo com que seja feito $2|V|^2$ operações.

A complexidade de espaço nesse algoritmo é dada pela própria instância do problema que precisa ser carregada na memória. Logo, a complexidade geral do algoritmo de busca por força bruta será:

Tabela 1. Complexidade da Força Bruta

Complexidade de Tempo	Complexidade de Espaço
$O(n^2)$	$O(n)$

3.2. Algoritmo Guloso

A heurística gulosa utilizada neste trabalho diminui drasticamente a complexidade total do algoritmo. Primeiramente, é necessário calcular “peso” para cada aresta, a partir do tamanho da união dos focos que os vértices desta aresta visitam, sendo a fórmula definida por $w(u, v) = \frac{1}{|R(u) \cup R(v)|}$. Este processo tem um custo linear nas arestas.

Enquanto o processo de atribuição de pesos é feito, a aresta que contém o menor peso é armazenada em uma variável separada, evitando assim o custo de se fazer uma ordenação das arestas.

Após a escolha da aresta de menor custo, seus vértices são adicionados na solução e então começa a busca verificando os vizinhos de cada extremidade dos vértices que já estão na solução. Todo este processo também tem um custo linear nas arestas.

A complexidade de espaço é atribuída à nova lista de valores que é criada, sendo a lista de pesos de cada aresta que é consultada durante todo o algoritmo. Logo, temos que a complexidade geral da heurística gulosa é:

Tabela 2. Complexidade da heurística gulosa

Complexidade de Tempo	Complexidade de Espaço
$O(n)$	$O(n)$

3.3. Programação Dinâmica

A complexidade do algoritmo para o paradigma dinâmico possui forte relação à topologia do grafo que está sendo trabalhado. Pela característica do grafo anel, a construção da matriz H fica limitado apenas há n^2 subgrafos possíveis, não tendo custo nenhum de verificação de conexidade do subgrafo gerado.

Entretanto, este algoritmo perde muito em relação à complexidade de memória, sendo o custo de memória no pior caso definido pela seguinte equação:

$$n \sum_{i=1}^n i = n \left(\frac{n(n+1)}{2} \right) \Rightarrow O(n^3) \quad (1)$$

Com esse custo de trabalhos na memória, o tempo de execução do algoritmo na prática pode ser pior que o custo teórico calculado. Na tabela abaixo, temos um exemplo de construção de uma solução à partir da instância dada como exemplo. Primeiramente é gerado todos os subgrafos de tamanho 1. A partir destes subgrafos, é feita uma união com o subgrafo vizinho para gerar todos os subgrafos de tamanho 2, e assim sucessivamente até que seja encontrada uma solução ótima.

Tabela 3. Exemplo de construção de uma solução

[6]	[4]	[1]	[3]	[7]	[2]	[5]
[6,4]	[4,1]	[1,3]	[3,7]	[7,2]	[2,5]	[5,6]
[6,4,1]	[4,1,3]	[1,3,7]	[3,7,2]	[7,2,5]	[2,5,6]	[5,6,4]
:	:	:	:	:	:	:

Tabela 4. Complexidade da Programação Dinâmica

Complexidade de Tempo	Complexidade de Espaço
$O(n^2)$	$O(n^3)$

4. Implementação dos algoritmos

Os algoritmos acima foram implementados na linguagem *Python* e estão anexos à este trabalho. Cada vértice é um objeto definido pela classe *Person*, onde é possível atribuir um identificador, lista de adjacência e lista de focos que visita.

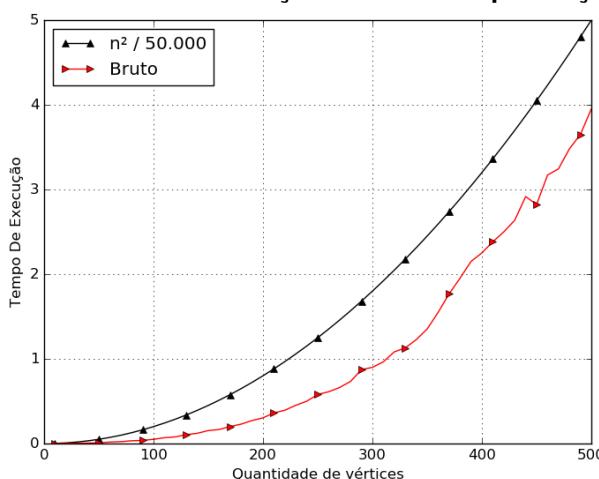
A classe *Problem* contém três métodos, sendo eles o *execute-brute*, *execute-greedy* e o *execute-dinamic*, onde de acordo com o paradigma escolhido o programa executará o método correto para iniciar a busca pela solução.

5. Execução dos Testes e Comparações

Os testes executados neste trabalho utilizaram 50 instâncias de pior caso, onde a entrada varia de 10 à 500 vértices. Logo abaixo, temos os gráficos de tempo de execução em relação ao tamanho da entrada. Junto aos gráficos, foram adicionadas as funções assintóticas que foram atribuídas na seção teórica, multiplicada por uma constante que mais se aproximou ao resultado obtido.

Na Figura 1 temos o gráfico de tempo de execução para o paradigma da busca por força bruta. Na seção teórica foi induzido que a complexidade deste algoritmo era $O(n^2)$, podendo ser confirmado ao comparar o aumento do tempo com a função $f(n) = \frac{n^2}{50000}$.

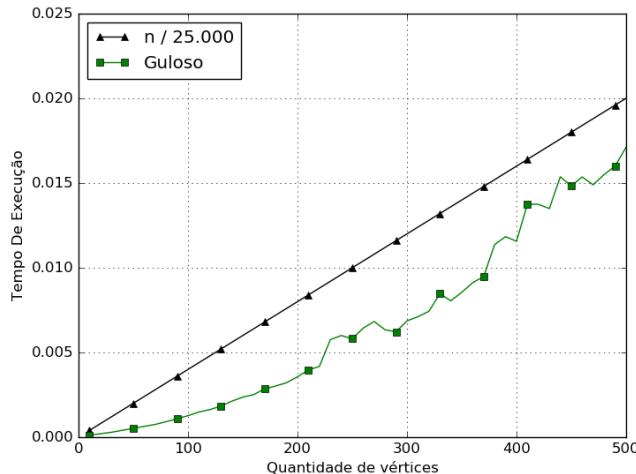
Figura 1. Gráfico de execuções com busca por força bruta



Na Figura 2 temos o gráfico de tempo de execução para o paradigma guloso. Na seção teórica foi induzido que a complexidade desta heurística era $O(n)$, podendo ser

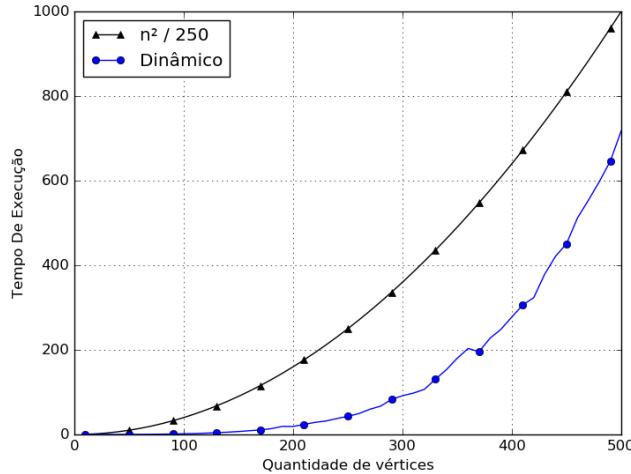
confirmado ao comparar o aumento do tempo com a função $f(n) = \frac{n}{25000}$. A distância da solução obtida com a solução ótima não foi verificado neste caso.

Figura 2. Gráfico de execuções com o algoritmo guloso



Na Figura 3 temos o gráfico de tempo de execução para o paradigma da programação dinâmica. Na seção teórica foi induzido que a complexidade deste algoritmo era $O(n^2)$ junto ao *overhead* de uso excessivo da memória. Entretanto, pode-se confirmar a complexidade ao comparar o aumento do tempo com a função $f(n) = \frac{n^2}{250}$.

Figura 3. Gráfico de execuções com o algoritmo de programação dinâmica



Na Figura 4, temos dois gráficos que ilustram a eficiência dos algoritmos implementados. No lado esquerdo, podemos observar que se não for necessário a solução ótima para o problema, o algoritmo guloso é o melhor à ser escolhido, dado seu desempenho linear.

Entretanto, no lado direto temos a comparação entre os algoritmos que encontram a solução ótima em todos os casos, a busca por força bruta e a programação dinâmica. Dado que o algoritmo de programação dinâmica faz um uso excessivo de memorização, o algoritmo por força bruta possui uma constante muito menor que faz com que sua esca-

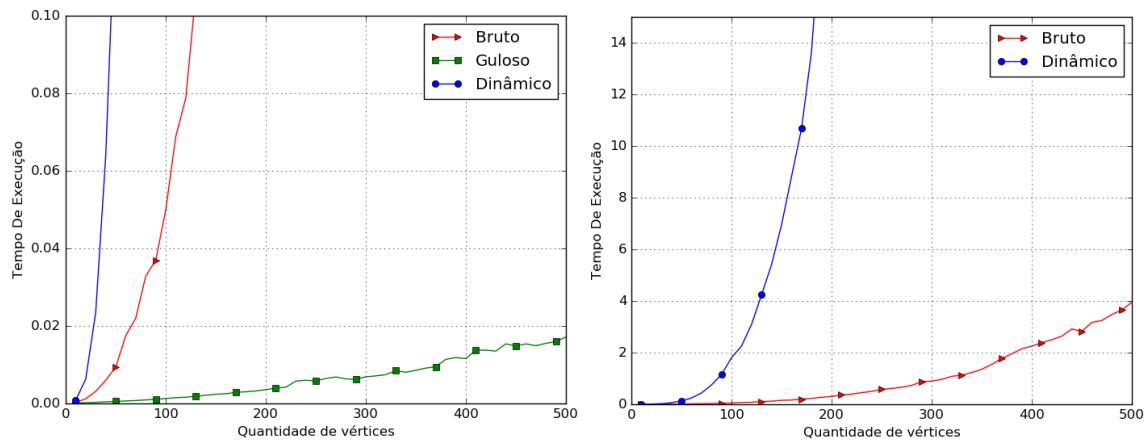


Figura 4. Eficiência do algoritmo guloso e a ganho da força bruta ao overhead de memória do dinâmico

labilidade seja mais efetiva que a programação dinâmica, sendo a melhor escolha como algoritmo para encontrar a solução exata.

6. Conclusão

Neste trabalho, fizemos um estudo do problema do ZicaZeroAnelDual, onde à partir de implementações de três algoritmos de diferentes paradigmas pudemos comparar a eficiência entre eles.

A heurística gulosa apesar de não retornar uma solução ótima, seu custo linear torna a solução bem interessante. Para algoritmos que buscam a solução ótima, o algoritmo de busca por força bruta se mostrou mais efetivo para **grafos suficientemente grandes**, entretanto para grafos pequenos ou que a sabe-se que a solução ótima será pequena, o algoritmo de programação dinâmica é o mais indicado para este caso.

Referências

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

Trabalho Prático: ZikaZeroAnelDual

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

O problema ZikaZeroAnelDual consiste em selecionar o menor número de voluntários $V' \subset V$, tal que, todo foco é acessado, por pelo menos um voluntário $v \in V'$, e o grafo induzido por V' em G_0 é conexo. Para isso, devem ser propostos algoritmos de complexidade assintótica polinomial com o tamanho da entrada (n, m e r) para resolver o Problema ZikaZeroAnelDual usando os seguintes paradigmas: Busca por força Bruta; Programação Dinâmica; e Algoritmo Guloso. Para resolver o problema proposto neste trabalho as abordagens consideraram que o grafo especificado no trabalho 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.

1.1. Busca por força Bruta

O algoritmo proposto baseado no paradigma de força bruta consiste em percorrer conjuntos C_i de vértices do grafo. Esses conjuntos serão compostos por uma quantidade variável de vértices, sendo essa quantidade aumentada gradativamente, de $1..n$, para $n = |V|$. Dessa forma, os conjuntos serão percorridos, considerando a quantidade inicial de vértices igual a 1. É verificado se os focos $F(v_i)$ que podem ser acessados pelo voluntário v_i são suficientes para realizar toda a cobertura do grafo. Caso seja encontrado algum vértice que consiga cobrir todos os focos do grafo, o algoritmo é parado e a resposta contendo qual vértice realizar tal cobertura é armazenada no arquivo de saída. Caso contrário, o conjunto irá conter o próximo vértice do grafo e realizar o mesmo procedimento de verificação. Este processo será repetido para cada vértice do grafo.

Normalmente, será necessário um número maior de vértices (voluntários) para realizar a cobertura dos focos, por isso a quantidade de vértices por conjunto será aumentada para 2. Com isso, o primeiro conjunto contendo os vértices v_1 e v_2 , será avaliado visando determinar se os focos $F(v_1)$ e $F(v_2)$ são suficientes para realizar a cobertura dos focos do grafo. Se os referidos vértices conseguirem realizar a cobertura de todos os focos do grafo, a resposta contendo quais vértices realizam a cobertura será escrita no arquivo de saída. Caso contrário, a próxima combinação de vértices v_2 e v_3 será considerada pelo conjunto e o mesmo procedimento de verificação será executado. Esse processo será repetido até que todas as combinações, possíveis, de dois vértices sejam consideradas pelo conjunto e verificadas.

A quantidade de vértices por conjunto será novamente incrementada permitindo agora a verificação de todas as combinações possíveis de três vértices. Os focos dos vértices serão verificados e caso seja encontrada uma solução contendo os vértices que realizam a cobertura, a execução é parada e a solução é armazenada no arquivo de saída.

Seguindo essa sequência, a quantidade de vértices por conjunto será incrementada até n , ou seja, até ter verificado todas as possíveis combinações de vértices que podem realizar a cobertura de todos os focos do grafo.

É importante destacar que os vértices que correspondem à solução do problema são organizados e armazenados em ordem crescente. Bem como, que a estrutura de resolução do problema garante que a solução com a menor quantidade de vértices é encontrada e com a menor soma dos vértices.

1.2. Algoritmo Guloso

O algoritmo proposto baseado no paradigma guloso também considera a ideia de conjuntos de vértices utilizada na abordagem baseada em força bruta. Para isso, deve ser definido um conjunto de vértices C , de modo que, a quantidade de vértices contidos no conjunto irá crescer gradualmente de $1..n$, para $n = |V|$.

Inicialmente o conjunto C irá conter um vértice (v_1), então é verificado se v_1 é capaz de cobrir todos os focos do grafo. Se o vértice realiza a cobertura dos focos, a execução é parada e a solução é armazenada no arquivo de saída. Caso contrário, será realizada uma verificação entre o vértice contido no conjunto (v_1) e os vértices adjacentes v_2 e v_n . Será avaliado qual dos vértices adjacentes pode ser agregado ao conjunto de forma a maximizar a cobertura dos focos do grafo. Ao constatar qual a solução ótima local o referido vértice é agregado ao conjunto. Se a solução para o problema for encontrada a execução é parada e a solução é armazenada no arquivo de saída. Se ao avaliar os elementos adjacentes foi constatado um empate, então os vértices adjacentes dos adjacentes são consultados como critério de tomada de decisão para determinar qual escolha realizar.

Após acrescentar o segundo elemento ao conjunto, é novamente realizada uma verificação entre os vértices contidos no conjunto (v_1 e v_n) e os vértices adjacentes v_2 e v_{n-1} , para determinar qual dos elementos adjacentes pode ser agregado ao conjunto de forma a maximizar a cobertura dos focos do grafo. Ao determinar qual a solução ótima local o referido vértice é agregado ao conjunto. Se a solução para o problema for encontrada a execução é parada e a solução é armazenada no arquivo de saída. Se ao avaliar os elementos adjacentes foi constatado um empate, então os vértices adjacentes dos adjacentes são consultados como critério de tomada de decisão para determinar qual escolha realizar. Esse processo é repetido até que o conjunto contenha todos os vértices do grafo ou até que a solução seja encontrada.

É importante destacar que os vértices que correspondem à solução do problema são organizados e armazenados em ordem crescente. Bem como, que a estrutura de resolução do problema consiste em um conjunto de tomadas de decisão que consideram somente o contexto local para a escolha da solução ótima local, sem ter qualquer conhecimento do contexto geral da aplicação.

2. Exercício 2

2.1. Busca por força Bruta

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: $O(V^2)$, 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 baseada em força bruta com complexidade $O(V^2)$, já que as combinações são verificadas sempre dois a dois (entre um novo vértice e o conjunto). A ordenação dos vértices que compõem a solução do problema apresenta complexidade $O(V \log V)$. Logo, analisando a complexidade de cada parte do algoritmo é possível observar que o cálculo das combinações de voluntários domina assintoticamente a complexidade do algoritmo que é de $O(V^2)$. Sendo importante destacar que, a estrutura da solução permite o encerramento do algoritmo ao encontrar a solução do problema, sendo possível obter um melhor desempenho na realização da tarefa. Analisando a complexidade espacial do algoritmo proposto, devem ser consideradas a matriz de adjacências, matriz de focos e o vetor que corresponde ao conjunto. A complexidade espacial da matriz de adjacências é $O(V^2)$, a complexidade espacial da matriz de focos é definida por $O(V.F)$ e a complexidade espacial do vetor conjunto é $O(F)$. Resultando em um complexidade espacial final de $O(V^2)$.

2.2. Algoritmo Guloso

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: $O(V^2)$, 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 gulosa com complexidade $O(V)$ pois cada vértice só é percorrido uma vez. A ordenação dos vértices que compõem a solução do problema apresenta complexidade $O(V \log V)$. Logo, analisando a complexidade de cada parte do algoritmo é possível observar que a inicialização das matrizes domina assintoticamente a complexidade do algoritmo que é de $O(V^2)$. Sendo importante destacar que, a estrutura da solução permite o encerramento do algoritmo ao encontrar a solução do problema, sendo possível obter um melhor desempenho na realização da tarefa. Analisando a complexidade espacial do algoritmo proposto, devem ser consideradas a matriz de adjacências, a matriz de focos e o vetor que corresponde ao conjunto. A complexidade espacial da matriz de adjacências é $O(V^2)$, a complexidade espacial da matriz de focos é definida por $O(V.F)$ e a complexidade espacial do vetor conjunto é $O(F)$. Resultando em um complexidade espacial final de $O(V^2)$.

3. Exercício 3

Os algoritmos propostos como solução do problema apresentado neste trabalho, foram implementados na linguagem C. O diretório Trab3-ZikaZeroAnelDual, contém os arquivos: `compilar-bruto.sh`, `compilar-guloso.sh`, `executar-bruto.sh`, `executar-guloso.sh`, `Bruto.c`, `Guloso.c`, os arquivos de entrada fornecidos pelo monitor da disciplina (`in` até `in5`) e os arquivos de entrada de 5 exemplos solicitados no Exercício 3 (`tst1` até `tst3`). Para a compilação e execução dos programas, como definido na especificação do trabalho, devem ser executados os arquivos `compilar-bruto.sh` e/ou `compilar-guloso.sh`, da seguinte forma: `./compilar-bruto.sh`. Em seguida, devem ser executados os arquivos `executar-bruto.sh` e/ou `executar-guloso` da seguinte forma: `./executar-bruto.sh in0 out0`. Onde será realizada a leitura do arquivo de entrada `in0` e será produzido um arquivo de saída `out0`, com a solução do problema para aquela configuração.

4. Exercício 4

Para experimentar a abordagem apresentada, foram propostos alguns exemplos. O primeiro exemplo é composto por 7 vértices, 7 arestas e 6 focos, como mostrado na Figura

1 abaixo. Para este exemplo foi apresentado o conjunto solução: 1, 2, 3 e 7.

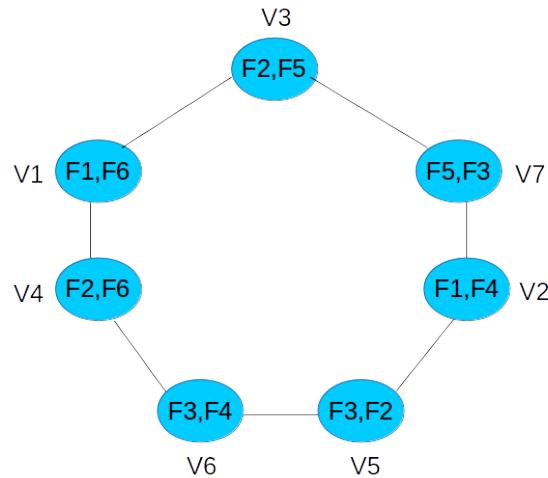


Figura 1. Exemplo 1.

Para o segundo exemplo, é apresentado um grafo com 10 vértices, 10 arestas e 12 focos, como pode ser visto na Figura 2. Para este exemplo é apresentado o conjunto solução: 3, 4, 5, 6, 8 e 9.

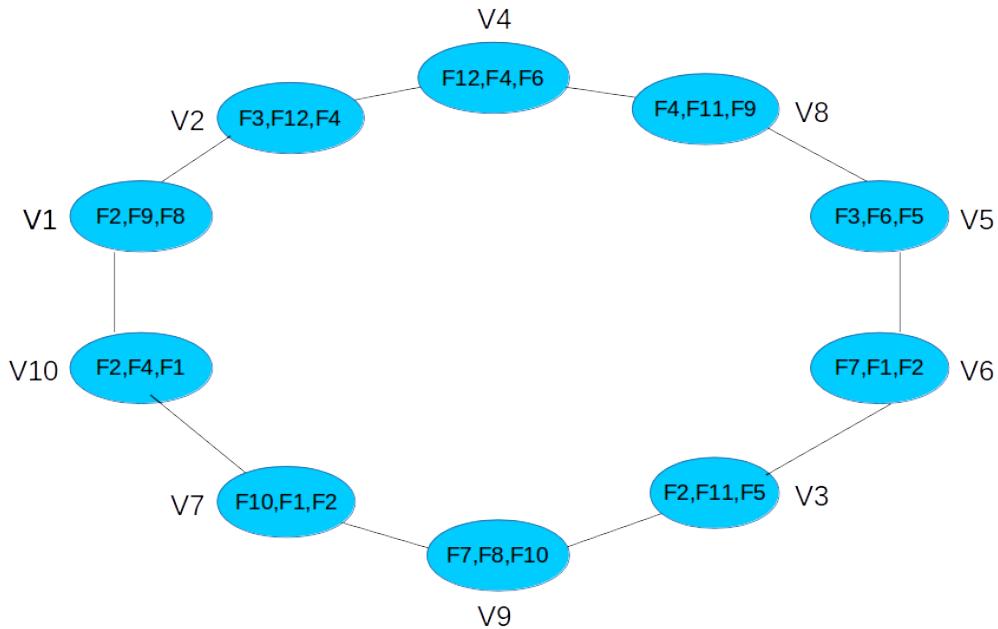


Figura 2. Exemplo 2.

No terceiro exemplo, é apresentado um grafo com 8 vértices, 8 arestas e 10 focos, como pode ser visualizado na Figura 3. Para este exemplo é apresentado o conjunto solução: 4, 5 e 6.

Para avaliar o desempenho temporal das abordagens proposta foram realizados experimentos variando o número de vértices, partindo de 1 até 50 e fixando a quantidade de arestas em função da quantidade de vértices (V) e a quantidade de focos em $2*|V|$.

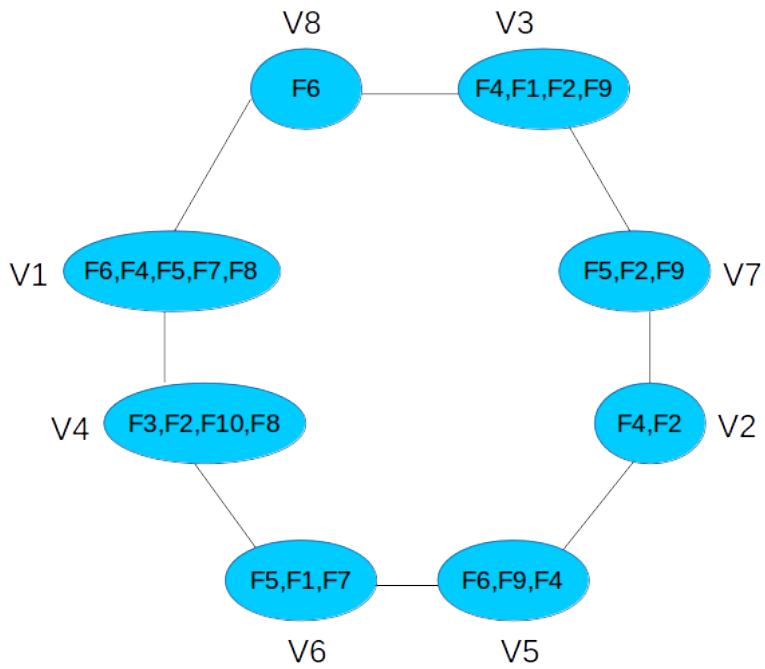


Figura 3. Exemplo 3.

Para cada variação de vértices, foram realizadas 10 execuções, sendo então tirada a média das 10 para representar o tempo de execução para um número específico de vértices. Na Figura 4 abaixo é possível observar o crescimento do tempo de execução para um número maior de vértices, com valor de focos fixado em 40 e uma variação de vértices em 20, 30, 40 e 50. Já que o cálculo das combinações é baseado no número de vértices, considerando a abordagem baseada em força bruta. Na Figura 5 abaixo é possível avaliar o crescimento do tempo de execução da abordagem baseada em força bruta quanto o número de focos é variado. Na Figura 5 foi fixado o número de vértices igual a 50 e foi variado o número de focos em: 1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100. Na Figura 6 é mostrado o baixíssimo crescimento do tempo de execução, da abordagem gulosa, quando o número de vértices é variado (20, 30, 40 e 50), com um valor fixo de focos de 40. Na Figura 7, é avaliado a taxa de crescimento do tempo de execução quando o foco é variado, considerando a abordagem gulosa. No experimento da Figura 7 foi fixado o número de vértices igual a 50 e foi variado o número de focos em: 1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100. Para a abordagem gulosa o tempo de execução A partir de 30 vértices o tempo de execução aumenta substancialmente, permitindo uma melhor visualização da disparidade entre as abordagens.

5. Exercício 5

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, como o algoritmo baseado em força bruta apresentava complexidade $O(V^2)$, que é o dobro da complexidade do algoritmo guloso $O(V)$, esperava-se que o mesmo tivesse uma curva de crescimento muito maior do que o algoritmo guloso, como pode ser visto nos gráficos apresentados nas Figuras 4, 5, 6 e 7. Durante a experimentação só foi possível visualizar uma diferença significativa entre os

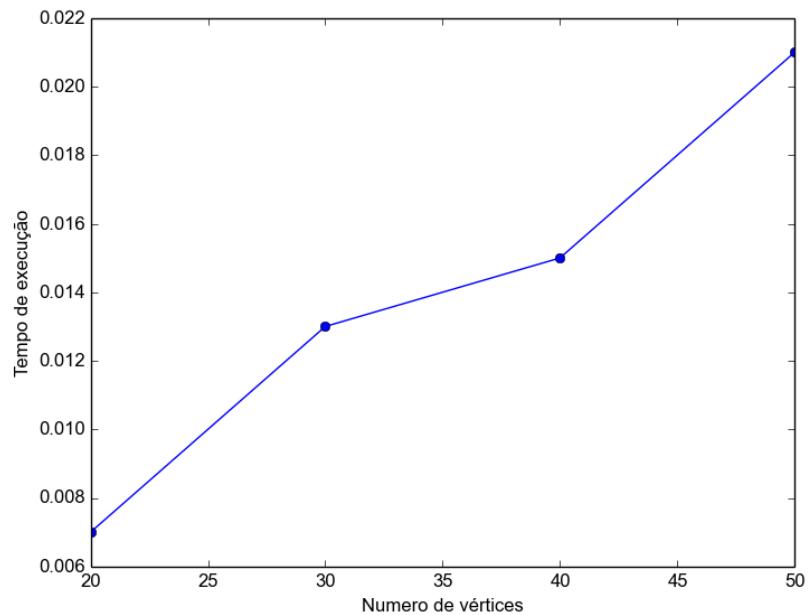


Figura 4. Gráfico 1 - Avaliação da abordagem baseada em força bruta, considerando variação do número de vértices.

tempos de execução dos algoritmos quando a quantidade de vértices foi aumentada para 30, já que para uma quantidade menor ambos executavam com uma taxa temporal muito semelhante.

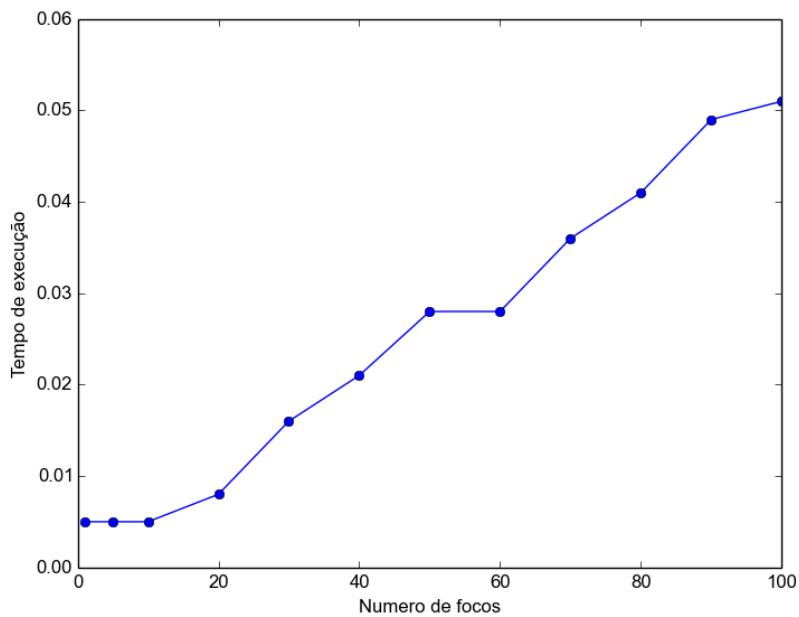


Figura 5. Gráfico 2 - Avaliação da abordagem baseada em força bruta, considerando variação do número de focos.

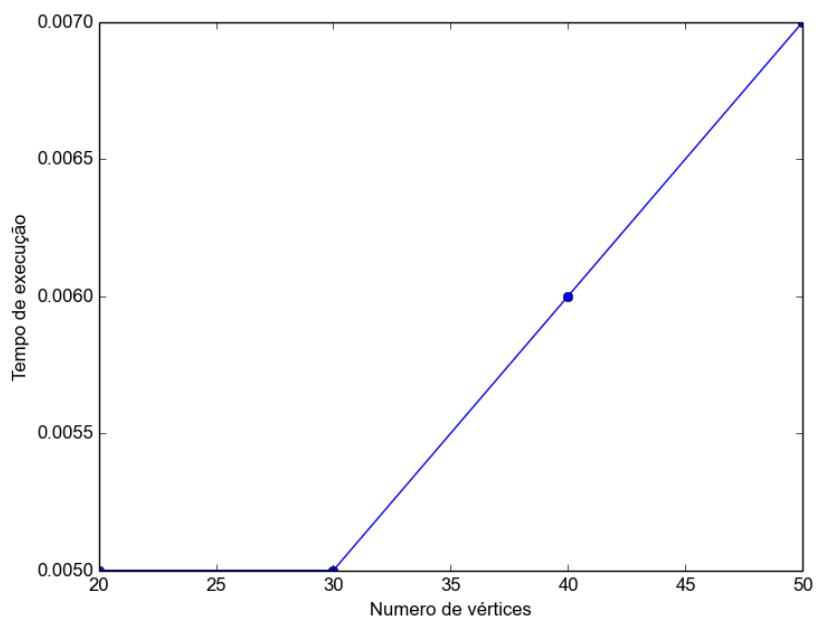


Figura 6. Gráfico 3 - Avaliação da abordagem gulosa, considerando variação do número de vértices.

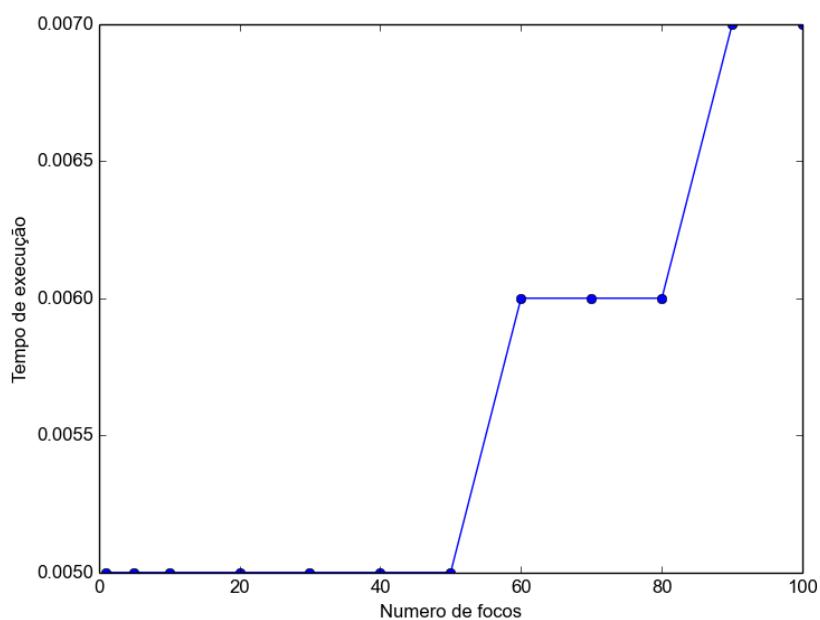


Figura 7. Gráfico 4 - Avaliação da abordagem gulosa, considerando variação do número de focos.

TP de Paradigmas - ZikaAnelDual

Thanis Paiva
Projeto e Análise de Algoritmos

17 de Junho de 2016

Exercício 1. Para resolver o problema ZikaAnelDual foram concebidos e implementados três algoritmos distintos que serão apresentados a seguir. Vale destacar que para todos os algoritmos, o grafo é armazenado em um vetor de uma dimensão e os focos cobertos por cada vértice em uma matriz de duas dimensões $n \times 2$. Assim, temos uma complexidade espacial inicial de $O(n) + O(2n) = O(n)$ para representar a estrutura do problema.

Além disso, todos os algoritmos usam a mesma estratégia de desempate, isto é, a partir de soluções com mesmo número de vértices, é retornado a solução com o menor somatório dos labels dos vértices. Caso o somatório seja o mesmo, é retornada a solução que apresentar o primeiro vértice distinto que seja menor.

Busca por Força-Bruta

Inicialmente, o algoritmo gera para cada vértice do grafo uma lista circular l_i que apresenta toda a sequência de vértices que o seguem até retornar a ele mesmo. Para o grafo da Figura 1, por exemplo, o vértice $V1$ tem a seguinte lista circular $l_1 = \{1, 3, 4, 2, 5, 1\}$, na qual cada vértice apresenta uma aresta com o vértice anterior. Cada lista l_i é percorrida e são geradas duas sublistas que cobrem todos os focos, uma no sentido horário, $l'_{início}$, que parte do primeiro vértice de l_i e caminha para a direita até cobrir todos os focos e uma no sentido anti-horário, l'_{fim} , que parte do último vértice de l_i e caminha para a esquerda até cobrir todos os focos. Por exemplo, para $l_1 = \{1, 3, 4, 2, 5, 1\}$, temos as sublistas $l'_{início} = \{1, 3, 4\}$ e $l'_{fim} = \{1, 5\}$. Dentre as $2n$ sublistas que cobrem todos os focos, seleciona-se a que apresenta menor número de vértices.

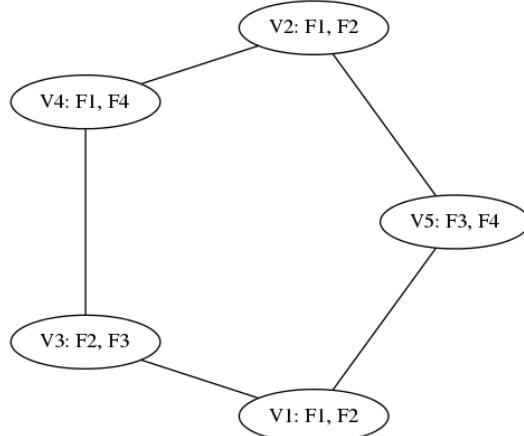


Figura 1: Exemplo de instância ZikaAnelDual, com $n = m = 6$ e $f = 4$

Algorithm 1 Algoritmo Força-Bruta

Require: $n \leftarrow numeroVertices$, $f \leftarrow numeroFocos$, $\mathbb{G} = (n, n)$, $\mathbb{F} \leftarrow \mathcal{R}(v)$

1: $\mathcal{L} \leftarrow listasCirculares$ ▷ Listas circulares de cada vertice
2: $\mathcal{S}_{cover} = \{\}$ ▷ Sublistas que cobrem todos os focos
3: $\mathcal{S}_{bruta} = \{\}$ ▷ Menores sublistas que cobrem todos os focos
4: **for each** $l \in \mathcal{L}$ **do**
5: $l'_{inicio} = \{\}, l'_{inicio_f} = \{\}$
6: $l'_{fim} = \{\}, l'_{fim_f} = \{\}$
7: $i \leftarrow 0$
8: **while** $l'_{inicio_f}.size \neq f$ **do**
9: $l'_{inicio} = l'_{inicio} \cup l[i]$
10: $l'_{inicio_f} = l'_{inicio_f} \cup focos(l[i])$
11: $i \leftarrow i + 1$
12: $\mathcal{S}_{cover} = \mathcal{S}_{cover} \cup l'_{inicio}$
13: $j \leftarrow n + 1$
14: **while** $l'_{fim_f}.size \neq f$ **do**
15: $l'_{fim} = l'_{fim} \cup l[i]$
16: $l'_{fim_f} = l'_{fim_f} \cup focos(l[i])$
17: $j \leftarrow j - 1$
18: $\mathcal{S}_{cover} = \mathcal{S}_{cover} \cup l'_{fim}$
19: $minimo \leftarrow MINIMO(\mathcal{S}_{cover})$
20: **for each** $l' \in \mathcal{S}_{cover}$ **do**
21: **if** $l'.size == minimo$ **then**
22: $\mathcal{S}_{bruta} = \mathcal{S}_{bruta} \cup l'$
23: **if** $\mathcal{S}_{bruta}.size \neq 1$ **then**
24: $\mathcal{S}_{bruta} \leftarrow DESEMPATA(\mathcal{S}_{bruta})$
25: **return** ORDENA(\mathcal{S}_{bruta})
26: **procedure** DESEMPATA(\mathcal{S})
27: **for** $i = 1 \dots \mathcal{S}.size$ **do**
28: $labels_1 \leftarrow SOMALABELS(S_i)$
29: $labels_2 \leftarrow SOMALABELS(S_{i+1})$
30: **if** $labels_1 == labels_2$ **then return** MENORESLABELS(S_i, S_{i+1})
31: **else**
32: **if** $labels_1 < labels_2$ **then return** S_i
33: **else return** S_{i+1}

Programação Dinâmica

O grafo é circular, assim dados os vértices $\{i, k, j\}$, partindo do vértice i , temos que calcular a lista de focos cobertos, l_{ik} , da sequência de i até k e depois a lista l_{ij} , da sequência de i até j . Para isso, podemos calcular l_{ik} e reusar esse cálculo para encontrar l_{ij} , que corresponde à l_{ik} acrescido de j . Percebendo essa característica do problema, o algoritmo desenvolvido constrói uma tabela $n \times n$ no qual cada célula $F(i, j)$ contém a lista de focos do caminho entre o vértice i até o vértice j e F_i corresponde a lista de focos acessados pelo vértice i . Assim, temos que:

$$F(i, j) = \begin{cases} F_i & \text{se } i = j(1) \\ F(i, j - 1) + [F_j \setminus [F_j \cap F(i, j - 1)]] & \text{se } j > i \text{ ou } 0 < j < i(2) \\ F(i, n - 1) + [F_j \setminus [F_j \cap F(i, n - 1)]] & \text{se } j = 0(3) \end{cases}$$

As linhas e colunas da tabela representam os vértices a partir do qual a sequência é gerada, isto é, a célula $F(i, j)$ armazena a lista de focos acessada pela sequência de vértices que começa em i e termina em j . Para isso, inicialmente o grafo passa por uma linearização, de forma que os índices $i, j = \{0, 1, 2, \dots, n - 1\}$, representem o primeiro vértice da linearização, o segundo vértice da linearização e assim por diante.

A tabela é preenchida linha por linha. A ordem de preenchimento, assim como as fórmulas usadas podem ser observadas na Figura 2a. Primeiramente calcula-se a diagonal da tabela (1) (vermelho), que apresenta um único vértice e consequentemente cobre apenas dois focos. Em seguida, para cada linha, calcula-se os valores após a diagonal (2) (azul), reusando os valores das células anteriores. Por fim, preenche-se a coluna 0 (3) reusando o último valor calculado em (2), pois temos uma lista circular, e calcula-se os valores de $j = 1$ até a diagonal usando a fórmula (2) (verde). Depois de preencher toda a tabela, temos todas as combinações de listas de focos possíveis.

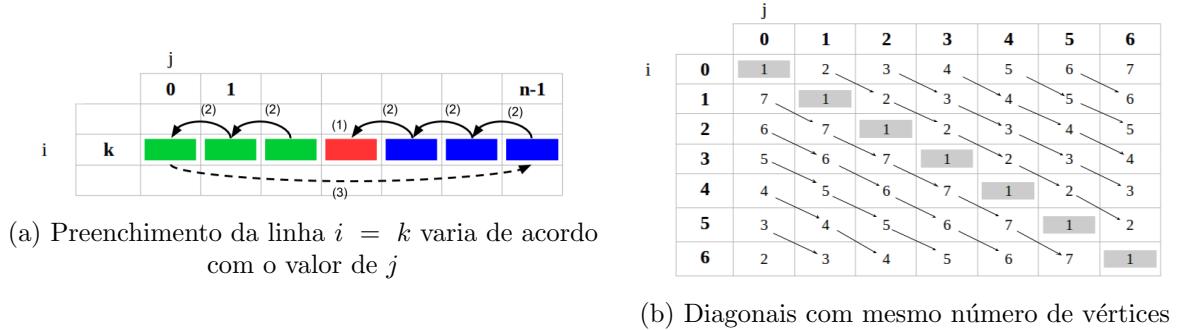


Figura 2: Tabela é calculada linha por linha e em seguida as diagonais são percorridas em busca da solução com número mínimo de vértices

Cada diagonal possui sequências de vértices com o mesmo número de vértices, como podemos observar na Figura 2b. Dessa forma, o algoritmo precisa percorrer a tabela nas diagonais acima e abaixo da diagonal principal, verificando primeiro as diagonais que possuem dois vértices, depois as que possuem três vértices e assim por diante, até encontrar as células que tem uma lista de focos que cobre todos os focos. Assim, que for encontrado uma célula que cobre todos os focos, terminamos de percorrer aquela diagonal e paramos de percorrer a tabela, pois todas as soluções seguintes apresentarão um maior número de vértices e desejamos obter uma solução com o mínimo de vértices. Todas as células encontradas na diagonal de menor

número de vértices tem suas sequências de vértices armazenadas como solução em potencial. Por fim, caso sejam identificadas soluções com o mesmo número de vértices, é retornada a que apresenta o menor somatório dos labels dos vértices.

Algorithm 2 Algoritmo Programação Dinâmica

Require: $n \leftarrow numeroVertices$, $f \leftarrow numeroFocos$, $\mathbb{G} = (n, n)$, $\mathbb{F} \leftarrow \mathcal{R}(v)$

```

1:  $L \leftarrow grafoLinearizado$                                  $\triangleright$  Lista de vertices que indexa a tabela
2:  $F[L][L]$                                                   $\triangleright$  Tabela de dimensões  $n \times n$ 
3: for  $i = 0 \dots n - 1$  do                                 $\triangleright$  Percorre a linha
4:   for  $j = 0 \dots n - 1$  do                             $\triangleright$  Percorre as colunas
5:     if  $i == j$  then                                 $\triangleright$  Diagonal, aplica (1)
6:        $F[i][j] = F_i$ 
7:     if  $j > i$  then                                 $\triangleright$  Depois da diagonal, aplica (2)
8:        $F[i][j] = F[i][j - 1] + [F_j \setminus [F_j \cap F[i][j - 1]]]$ 
9:     if  $j < i$  then                                 $\triangleright$  Antes da diagonal, aplica (3) e (2)
10:    if  $j == 0$  then                                 $\triangleright$  1a coluna, aplica (3)
11:       $F[i][n - 1] = F[i][n - 1] + [F_j \setminus [F_j \cap F[i][n - 1]]]$ 
12:    else
13:       $F[i][j - 1] = F[i][j - 1] + [F_j \setminus [F_j \cap F[i][j - 1]]]$      $\triangleright$  apos 1a coluna, usa (2)
14: for all  $diagonal \in F[L][L]$  do
15:   for all  $diagonalMesmoTamanho \in F[L][L]$  do
16:     if  $|F[i][j]| == f$  then                                 $\triangleright$  Achou diagonal com menos vertices
17:        $S_{cover} \leftarrow SEQUENCIA(i, j)$ 
18:     if  $S_{cover}.size \neq 1$  then
19:        $S_{cover} \leftarrow DESEMPATA(S_{cover})$ 
20:      $S_{dinamico} \leftarrow ORDENA(S_{cover})$ 
21: return  $S_{dinamico}$ 

```

Algoritmo Guloso

Inicialmente é escolhido um vértice de origem a partir do qual a solução será crescida. Para isso, o algoritmo percorre cada vértice e calcula o seu custo inicial. O custo inicial é a quantidade de focos cobertos pelo vértice i que também são cobertos por um dos seus vértices adjacentes. Assim o custo assume três valores considerando que cada vértice acessa dois focos distintos:

$$Custo(i) = \begin{cases} 0 & \text{se nenhum dos focos de } i \text{ são cobertos pelos seus adjacentes (sem repetição)} \\ 1 & \text{se apenas um dos focos de } i \text{ é coberto pelos seus adjacentes (um se repete)} \\ 2 & \text{se ambos os focos de } i \text{ são cobertos pelos seus adjacentes (dois se repetem)} \end{cases}$$

Seleciona-se como primeiro vértice o que apresentar menor custo, ou seja, o que tem a vizinhança imediata formada pelos dois adjacentes, com o maior número de focos distintos. Em outras palavras, o custo inicial mede o número de conflitos, ou focos repetidos, de um vértice i quando consideramos os seus dois vértices adjacentes. Quando menor o custo, maior o número em potencial de focos distintos cobertos pelo vértice i e um ou ambos os seus vértices adjacentes. Dessa forma, temos uma maior chance de reduzir o número de vértices necessários para a solução, se começarmos a partir de um vértice que tem adjacentes com os quais já são cobertos mais focos distintos. Assim, ordena-se os vértices pelo custo inicial, e é selecionado como origem o que apresentar menor custo e menor label.

Depois que o primeiro vértice foi escolhido, o algoritmo cresce a solução, adicionando em cada passo um vértice adjacente aos vértices do conjunto solução S , que apresente menor custo, ou seja, que adicione mais focos não-cobertos pela solução corrente. Assim, temos que o conjunto de candidatos C em cada iteração tem tamanho $|C| = 2$, e possui os vértices que são adjacentes aos vértices das extremidades do conjunto solução S . Calcula-se o custo de cada candidato e o que apresentar menor custo, ou seja, o que adiciona mais focos distintos é adicionado à S . Como as extremidades de S , crescem, em cada iteração temos um novo C e novos custos. Em caso de empate, é selecionado o vértice de menor label.

Algorithm 3 Algoritmo Guloso

Require: $n \leftarrow \text{numeroVertices}$, $f \leftarrow \text{numeroFocos}$, $\mathbb{G} = (n, n)$, $\mathbb{F} \leftarrow \mathcal{R}(v)$

- 1: $S \leftarrow \{\}$ ▷ Conjunto Solucao
- 2: $S_f \leftarrow \{\}$ ▷ Focos cobertos pela solucao
- 3: $\text{origem} \leftarrow \text{ACHAMENORCUSTOINICIAL}(\mathbb{G})$
- 4: $S \leftarrow \text{origem}$
- 5: $S_f \leftarrow \text{focos}(\text{origem})$
- 6: $C \leftarrow \{\}$ ▷ Vertices candidatos
- 7: $C_{\text{custo}} \leftarrow \{\}$ ▷ Custos dos candidatos
- 8: **while** $S_f.size \neq f$ **do**
- 9: $C \leftarrow \text{ADJACENTES}(S)$
- 10: $C_{\text{custo}} \leftarrow \text{ATUALIZACUSTOS}(S_f, C)$
- 11: $\text{escolhido} \leftarrow \text{CUSTOMINIMO}(C_{\text{custos}})$ ▷ Retorna o menor label para mesmo custo
- 12: $S \leftarrow S \cup \text{escolhido}$
- 13: $S_{\text{guloso}} \leftarrow \text{ORDENA}(S)$
- 14: **return** S_{guloso}

Exercício 2. Em todos os algoritmos, o grafo é armazenado em um vetor de uma dimensão e os focos cobertos por cada vértice em uma matriz de duas dimensões $n \times 2$. Assim, temos uma complexidade espacial inicial de $O(n) + O(2n) = O(n)$ para representar a estrutura do problema. Temos um custo $O(n)$ para percorrer o grafo e $O(1)$ para acessar os focos de um vértice. A seguir analisamos a complexidade temporal e espacial dos algoritmos considerando as computações e estruturas adicionais de cada paradigma.

Busca por Força-Bruta

Inicialmente, o algoritmo gera para cada vértice uma lista circular l_i , partindo de i e passando por todos os n vértices até voltar em i , assim, para qualquer entrada temos uma complexidade de $n \times O(n+1) = O(n)$ para gerar as listas circulares. Em seguida, cada lista l_i é percorrida duas vezes, uma partindo do início até o final da lista (1 até n) e outra do final até o início ($n+1$ até 2). Cada lista é gerada com custo $O(n)$ e temos duas sublistas por vértice, logo, com $2n$ sublistas, temos um custo total de $2n \times O(n) = O(n^2)$ para gerar todas as sublistas que cobrem todos os vértices. As sublistas são percorridas para identificar a que apresenta menor número de vértices, a um custo de $O(2n) = O(n)$ e ela é retornada. Em caso de empate, é seguida a estratégia de solução com menor soma de labels. Logo temos uma complexidade polinomial de $O(n^2)$.

Em termos de espaço, além da estrutura de armazenamento que é comum a todos os algoritmos, usamos n vetores com $n+1$ vértices para armazenar as listas circulares, e armazenamos dentre as $2n$ sublistas, apenas as que cobrem todos os focos. Logo, no pior caso, temos um custo de espaço de $O(n(n+1)) + O(2n) = O(n^2)$.

Programação Dinâmica

A tabela é preenchida linha por linha e cada célula armazena a lista de focos cobertos pela sequência de vértices que vai do vértice i até o vértice j . Como temos n vértices, temos n linhas e n colunas, o preenchimento de cada linha corresponde a percorrer todo o grafo partindo de i , passando pelos demais vértices e voltando para i . Em cada célula, adicionamos até dois focos na lista de focos cobertos com custo $O(1)$. Assim, para computar uma única linha teremos um custo de $n \times O(1) = O(n)$. Como temos n linhas, o custo total para computar a tabela é $n \times O(n) = O(n^2)$. Em termos de espaço, temos um custo de $O(n^2f)$, já que teremos n^2 células que no pior caso armazenam uma lista de focos de tamanho máximo f (todos os focos).

Uma vez que a tabela foi computada, devemos percorrer as diagonais com o mesmo número de vértices, começando das diagonais com menor número de vértices em busca da diagonal que apresenta pelo menos uma célula cuja lista de focos cobre todos os focos. No pior caso, teremos que checar a cobertura de todas as células da tabela, a um custo de $O(n^2)$. Dessa forma, temos uma complexidade final de $O(n^2) + O(n^2) = O(n^2)$ para o algoritmo.

Guloso

Inicialmente calculamos o custo inicial de todos os n vértices para selecionar o vértice de origem. Para isso todo o grafo é percorrido a um custo de $O(n)$. Os custos são ordenados a um custo de $O(nlogn)$ e seleciona-se a origem. Durante as iterações do algoritmo em cada passo temos $|C| = 2$ candidatos que são adjacentes às extremidades com conjunto solução S , calculamos o custo de cada um fazendo a interseção com os focos já cobertos por S , a um custo constante. No pior caso todos os n vértices são adicionados, isto é, temos $n - 1$ iterações para crescer a solução S que já começa com um vértice, logo teremos um custo de $O(n) + O(nlogn) + (n - 1) \times O(1) = O(nlogn)$.

Em termos de complexidade de espaço, temos três vetores de uma dimensão, um armazena os n custos de cada vértice, um armazena os vértices de solução e o outro os focos acessados. Assim, temos uma complexidade de espaço de $O(n) + O(n) + O(f) = O(n)$.

Exercício 4. Foi realizado um experimento com o objetivo de comparar os tempos de execução dos algoritmos implementados. Foram realizadas 30 execuções para cada arquivo de teste em cada paradigma e o tempo de execução foi medido em milissegundos. Os arquivos de teste foram gerados de forma aleatória, com $n = \{10, 20, \dots, 100\}$ vértices e número máximo de focos, ou seja, com $f = 2n$. Ao definir para os arquivos de teste o número máximo de focos, garantimos que todos os vértices tenham que fazer parte da solução e consequentemente conseguimos atingir o pior caso para todos os algoritmos implementados. A média dos tempos de execução obtidos são evidenciados no gráfico da Figura 3.

Exercício 5. Podemos observar pela Figura 3, que o tempo de execução do algoritmo de força-bruta e programação dinâmica apresentam tempos de execução muito próximos. Isso ocorre pois o força-bruta implementado não gera todas as combinações de vértices existentes, de forma que já era esperado que seu tempo de execução fosse similar ao tempo de execução do algoritmo de programação dinâmica. O algoritmo de força-bruta, gera para cada vértice i apenas duas listas de vértices que cobrem todos os focos, partindo de i e percorrendo o grafo no sentido horário e anti-horário, até voltar em i ou até cobrir todos os focos. Com isso, conseguimos gerar os subgrafos conexos que cobrem todos os vértices com um custo polinomial. Após a geração das $2n$ listas, elas são percorridas para selecionar a menor lista que cobre todos os focos.

No caso do algoritmo de programação dinâmica, também são calculadas todas as com-

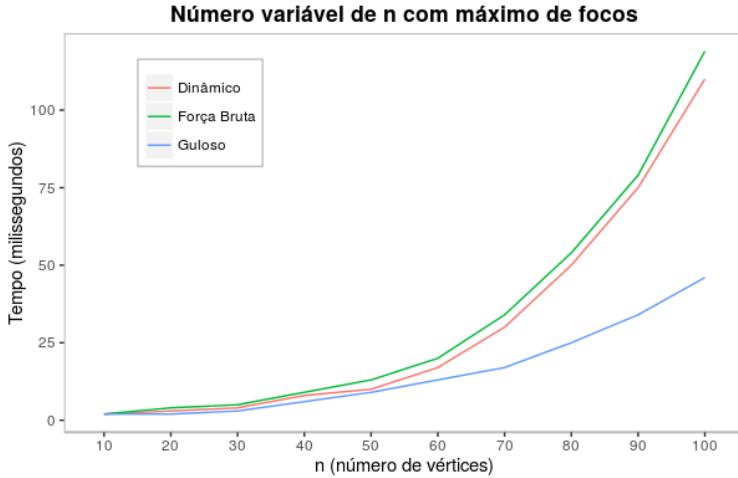


Figura 3: Comparação do tempo de execução entre os algoritmos de Força-Bruta, Programação Dinâmica e Guloso.

binações entre vértices sequenciais do grafo que geram subgrafos conexos. Porém, na hora de buscar a solução não precisamos percorrer todas as linhas, mas apenas as diagonais. Entretanto, no pior caso, todos os vértices fazem parte da solução, e com isso toda a tabela é percorrida. Logo, temos tempos de execução similares para o algoritmo de força-bruta e de programação dinâmica.

É evidente que o algoritmo guloso é o mais eficiente entre os algoritmos desenvolvidos. O seu maior custo é o de calcular o custo inicial dos vértices e ordená-los para selecionar o vértice de origem. Depois dessa etapa de custo $O(n \log n)$, é possível crescer a solução adicionando um vértice em cada iteração, logo no pior caso adicionamos $n - 1$ vértices, com custo $O(n)$, já que temos custo constante para calcular o custo dos dois candidatos e selecionar o próximo vértice a ser incluído na solução. Logo, o passo de ordenação é o mais custoso do algoritmo guloso, que apresenta assim, um tempo de execução inferior às outras abordagens, já que não calcula todas as sequências de vértices que levam a um grafo conexo, calculando apenas uma sequência que é dada como resposta. Porém, apesar de ser mais rápido, temos que o algoritmo guloso não retorna a solução ótima, ao contrário dos algoritmos de força-bruta e programação dinâmica.

A partir do experimento realizado foi possível comparar a execução dos algoritmos de força-bruta, programação dinâmica e guloso implementados. Na análise teórica realizada, encontramos que os algoritmos implementados de força-bruta e programação dinâmica apresentam o mesmo limite superior de $O(n^2)$, já o guloso apresenta complexidade de $O(n \log n)$. Assim, era esperado que o algoritmo de busca por força-bruta e o de programação dinâmica apresentassem tempos de execução similares e que os tempos de execução do algoritmo guloso fossem inferiores a ambos. Dessa forma, temos que a análise teórica foi compatível com o experimento realizado.

Trabalho Prático: ZikaZeroAnelDual

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 três soluções, utilizando força bruta, programação dinâmica e gulosa, para encontrar o menor grupo de voluntários que abrangem todas as áreas de focos.

1. Exercício 1 – Proponha algoritmos de Complexidade Assintótica Polinomial com o Tamanho da entrada (n , m e r) para resolver o Problema ZikaZeroAnelDual usando os seguintes paradigmas.

Para os algoritmos propostos abaixo, considere $|V| = N$, $|E| = M$ e $|R| = F$.

1.1. Busca por Força-bruta

O algoritmo de força bruta irá testar todas as configurações possíveis para criar o conjunto solução. Como enunciado pelo problema *ZikaZeroAnelDual*, a solução é o menor conjunto de vértices adjacentes (condição necessária para que o grafo seja conexo) tal que o conjunto desses vértices cobrem todos os focos de uma instância. Em caso de empate, ou seja, se existem duas soluções validas com a mesma quantidade de vértices, a solução ótima será a que possuir a menor soma desses elementos. O Algoritmo 1 abaixo é o esquema do algoritmo proposto.

Para uma instância genérica $G=\{v1, v2, v3, v4, ..., vn\}$ ele cria uma árvore do espaço solução inicialmente vazio, após isso, adiciona os N vértices do grafo. Para cada vértice adicionado, ele verifica se essa configuração satisfaz o problema. Caso seja satisfeita, ele adiciona essa configuração atual como solução parcial do problema. Como queremos que em caso de empate, isto é, caso existam mais de uma configuração com a mesma quantidade de elementos, ele escolherá a configuração que possua a menor soma dos elementos, para isso, precisamos avaliar todas as configurações possíveis que existem até o mesmo nível dessa configuração na árvore T e substituir a solução parcial com a melhor caso exista.

Caso não seja encontrado uma solução para uma configuração atual, para cada nó atual serão adicionados x folhas (onde x é a quantidade de adjacentes novos da folha atual). Como o grafo é sempre dado em forma de anel, o único caso em que um nó possui dois vértices vizinho novos é quanto se trata de um nó no primeiro nível da árvore solução, para os demais casos, haverá no máximo um novo vértice. A Figura 1 abaixo simplifica o que foi dito.

Como pode ser visto, é um algoritmo recursivo que inicialmente passa-se como parâmetro, o grafo, um conjunto parcial de solução (inicialmente é vazio), um inteiro que

Data: $G = (V, A)$, inteiro i , lista $Partial$, lista V'

Result: V' tal que $V' \subseteq V$

begin

i=*i*+1;

if *valida*(G , *partial*) **then**

if V' não contém uma solução melhor que a *Partial* **then**

 | adiciona *partial* em V' ;

end

 return;

end

else

if *partial* ainda é vazio **then**

for $v \in V$ **do**

 | *partial* = { v };

 | *forca_bruta*(G , *i*, *partial*, V');

end

end

else

 | $v_1, v_2 \rightarrow$ adjacentes do ultimo elemento adicionado em *Partial*;

if v_1 não está ainda em *Partial* **then**

 | *Partial*1 = *Partial* \cup v_1 ;

 | *forca_bruta*(G , *i*, *Partial*1, V');

end

if v_2 não está ainda em *Partial* **then**

 | *Partial*2 = *Partial* \cup v_2 ;

 | *forca_bruta*(G , *i*, *Partial*2, V');

end

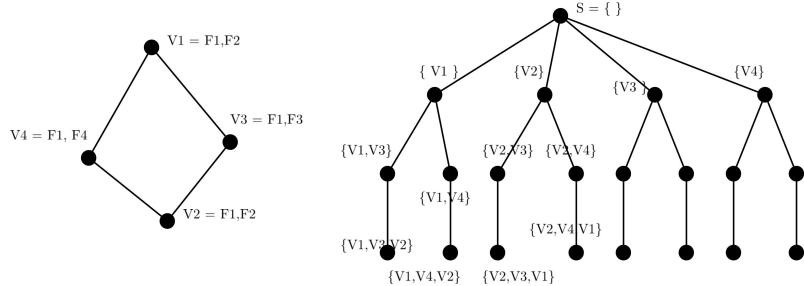
end

end

end

Algoritmo 1: Busca por Força-Bruta

Figure 1. Exemplo de criação da árvore do espaço solução



representa o nível da árvore (inicialmente é zero), e o conjunto de solução final (inicialmente vazio).

1.2. Programação Dinâmica

A solução para o problema no paradigma de programação dinâmica explora a propriedade que os focos cobertos por um ou mais vértices continuarão a serem cobertos ao adicionar mais um novo vértice no conjunto, além disso, a nova quantidade de focos cobertos ao adicionar um novo vértice será no mínimo igual à configuração anterior.

Uma outra propriedade é que com exceção ao caso onde a solução ótima é o conjunto de todos os vértices, a solução ótima será sempre na forma de um grafo na forma de linha. Esse fato é importante pois a cada novo vértice adicionado nos subproblemas só é possível adicionar mais um vértice adjacente a este (pois o outro já foi escolhido). No caso em que a solução ótima contenha todos os vértices, ao adicionar o último vértice, não será possível a adição de um novo adjacente, pois todos já foram anteriormente adicionados.

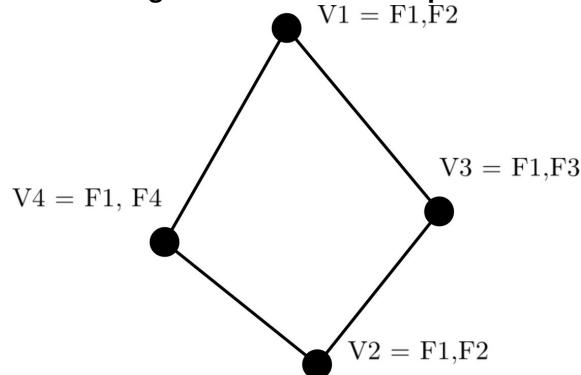
Considerando uma matriz $M[2^N][F]$, onde as i primeiras ($i < N$) linhas da matriz representam um vértice e de tal forma que a próxima linha representa necessariamente é um vértice adjacente. Considere ainda que a primeira linha é adjacente à N -ésima linha dessa mesma matriz e cada coluna j dessa matriz corresponde a 1 se o vértice cobre o foco j ou 0 senão cobre. As demais j ($j > N$) linhas da Matriz representam as soluções parciais em que se tem como vértice inicial o vértice da linha i tal que $i=j-N$.

Inicialmente, as j linhas ($j > N$) representam as soluções parciais onde se tem apenas um vértice, após uma atualização completa, representará as soluções parciais quando se adiciona o próximo vértice vizinho (totalizando dois vértices), e assim por diante. A Figura 2 e o trecho abaixo representam esse processo.

A partir desse grafo, temos 4 vértices e 4 focos distintos. A primeira posição da Matriz (Matriz[0]) irá conter os focos do vértice $v1$, a segunda linha irá conter os focos de $v3$ (pois é vizinho de $v1$), a terceira linha representará $v2$ e a quarta linha o vértice $v4$. Essa correspondência entre a numeração da linha e o vértice no grafo é salvo em uma tabela de tamanho N . Inicialmente, na primeira iteração do *loop*, adiciona-se cada vértice i na linha $i+N$ da Matriz, isso representa uma possível solução que inicia com o vértice i , a cada próxima iteração vai se adicionando o proximo vértice vizinho em cada possível solução. No exemplo abaixo, na primeira iteração, linha Matriz[4], inicialmente representa uma solução $\{v1\}$, já na segunda iteração $\{v1, v3\}$ e assim por diante.

Para $N=4$ e $F=4$

Figure 2. Grafo de exemplo



então na primeira
iteração temos:

Matriz[0] = [F1 F2]	Matriz[0] = [F1 F2]
Matriz[1] = [F1 F3]	Matriz[1] = [F1 F3]
Matriz[2] = [F1 F2]	Matriz[2] = [F1 F2]
Matriz[3] = [F1 F4]	Matriz[3] = [F1 F4]
Matriz[4] = Matriz[0]	Matriz[4] = Matriz[0] or Matriz[1]
Matriz[5] = Matriz[1]	Matriz[5] = Matriz[1] or Matriz[2]
Matriz[6] = Matriz[2]	Matriz[6] = Matriz[2] or Matriz[3]
Matriz[7] = Matriz[3]	Matriz[7] = Matriz[4] or Matriz[0]

A cada atualização é verificado se essa nova configuração corresponde a uma solução satisfazível. Como é necessário em caso de empate, garantir que a soma dos elementos seja a menor, é necessário avaliar também as próximas configurações até todas as linhas sejam atualizadas desse estágio.

O Algoritmo 2 simplifica as tarefas a serem feitas para chegar a uma solução a partir da programação dinâmica.

1.3. Algoritmo Guloso

Para esse problema, o algoritmo de guloso desenvolvido cria para simplificação das operações uma lista $G_ring[N]$, cada elemento na lista representa um vértice de tal forma que o próximo seja um vértice adjacente aos do lado.

O algoritmo começa procurando pelo primeiro vértice para fazer parte da solução, para isso, é feito uma leitura linear na lista G_ring e verificado quantos focos já foram cobertos, no momento em que com a leitura de um novo vértice, satisfaça a cobertura, significa que esse vértice contém focos que são importantes para a solução, logo, ele será o primeiro vértice a ser escolhido.

Apos a escolha do primeiro vértice, o algoritmo irá incrementar o conjunto solução, ate que todos os focos sejam cobertos. O processo de incrementar a solução é simples, inicialmente atualiza os valores da lista G_ring para -1 em todos os vértice que já foram escolhidos. E apos isso, descobre sempre dois novos candidatos, o primeiro candidato (vértice não escolhido) que esta a esquerda dos elementos que fazem parte da

Data: $G = (V, A)$
Result: V' tal que $V' \subseteq V$

```

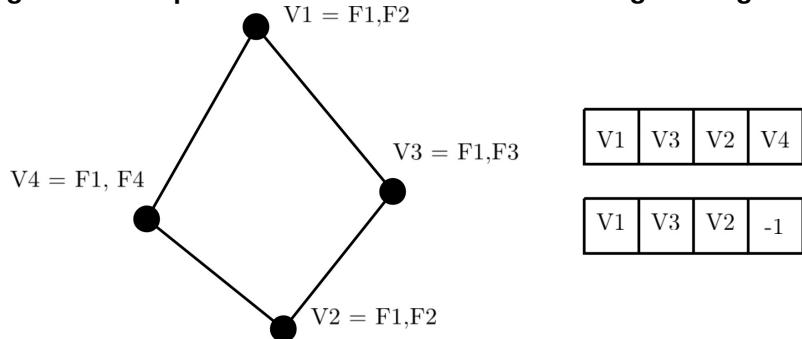
begin
    N = |V|;
    F = |F|;
    cria uma matriz de inteiros Matrix[2*N][F];
    cria uma matriz de inteiros Table[N];
    v = v1; i = 0;
    // Parte de construção inicial da matriz
    do
        Table[i]=v;
        for cada foco f do vértice v do
            Matrix[i][f]=1;
            Matrix[i+N][f]=1;
            i=i+1;
        end
        v = vértice adjacente de v não percorrido;
    while v ≠ NULL;
    i = N; achou = False; carry = 1; cc = 0;
    // Parte de construção da solução
    for i < N*N do
        for cada foco j possível de F do
            Matrix [cc+ N][j] = Matrix [cc+ N][j] or Matrix [ (carry + i)%N ][j];
        end
        cc+=1;
        if cc == N then
            cc =0; carry+=1;
            if achou == True then
                break;
            end
        end
        if Matrix[cc+ N] não contém nenhum foco zero then
            v_sol = cc; size_sol = carry; achou = True;
            adiciona [v_sol, size_sol] em solution se não houver solução melhor;
        end
        i=i+1;
    end
    v= v_sol em solution;
    size = size_sol em solution;
    i = 0; V' = ∅;
    for i < size_sol do
        V' = V' ∪ table[v_init];
        i=i+1;
    end
    ordena os elementos de V';
end

```

Algoritmo 2: Programação Dinâmica

solução, e o primeiro vértice a direita dos elementos que fazem parte da solução. A Figura 3 exemplifica o processo, isto é, primeiramente cria-se o vetor G_ring , depois o percorre (de $v1$ até $v4$), até encontrar uma solução (quando alcança o $v4$) e o adiciona na solução parcial.

Figure 3. Exemplo da escolha dos candidatos no algoritmo guloso.



Após isso, verificará qual dentre esses dois candidatos é o ótimo local, ou seja, qual dos dois vértices possuem mais focos que irão contribuir para a solução, e após isso, o adiciona. Em caso de empate, escolhe o vértice de menor valor.

Como dito, esse processo continua até que se encontre uma solução viável. No entanto, não se tem garantia de otimalidade de solução e também não tem um valor para se calcular a aproximação. O Algoritmo 3 enumera o passo-a-passo do que foi dito.

2. Exercício 2 – Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) dos algoritmos propostos no Exercício 1.

2.1. Busca por Força-bruta

A ordem de complexidade temporal e espacial do algoritmo por Força-Bruta é dado pelo pior caso, isto é, quando todos os vértices possuem focos que são únicos. Nesse caso, serão criadas $N^*2N + 1 = 2N^2 + 1$ chamadas recursivas e cada uma dessas chamadas farão as suas determinadas operações.

Em cada chamada recursiva, irá testar se a sua solução parcial é uma solução final na função $valida()$, isso exige uma complexidade temporal de $O(|F|)$. Também em cada chamada recursiva irá verificar se a solução parcial é vazia, essa verificação é $O(1)$ e só será válida uma vez em toda a execução, quando válida, será executado um laço N vezes. Se a condição não for válida, será procurado os dois vértices adjacentes do último vértice adicionado na lista parcial, como o grafo foi modelado em forma de lista de adjacências, obter os dois vértices gasta um tempo constante $O(1)$. Por fim, é verificado se esses vértices estão contidos já na lista e se não, o adicione na lista, isso tudo é feito em $O(N)$, pois no pior caso, será percorrido uma lista que contenha quase todos os elementos.

Antes de retornar a solução final de todo o processo, o algoritmo deverá ordenar a solução em ordem crescente, para o pior caso, gasta-se $O(N \log N)$.

Fica claro, então que a ordem de complexidade temporal será dada por $O((2N^2 + 1)(F + N + N + N) + N \log N) = O(N^2 F + N^3 + N \log N)$. Sabe-se também que pela definição do problema $F \leq 2N$, logo, para grandes quantidades de vértices, podemos ver que será da ordem de $O(N^3)$.

Data: $G = (V, A)$
Result: S tal que $S \subseteq V$

begin

```

G_ring[N] = criação da lista de vértices adjacentes;
focos[F];
v_first = -1;
S = {};
focos_v[2];
//escolha do primeiro vértice
for  $v \in G\_ring$  do
    | focos_v = pega os focos de v;
    | for  $f \in focos_v$  do
    |   | focos[f] = 1;
    | end
    | if cobriu todos os vertice then
    |   | v_first = v;
    |   | break;
    | end
end
S = S  $\cup$  v_first;
atualiza  $focos[F]$  com os apenas os focos cobertos por v_first;
//escolha dos próximos vértices
while existe um foco não coberto do
    | G_ring_tmp = G_ring;
    | for  $v$  in  $G\_ring$  do
    |   | if  $v$  esta em  $S$  then
    |   |   | G_ring_tmp = -1;
    |   | end
    | end
    | for  $v \in G\_ring$  do
    |   | if  $v == -1$  then
    |   |   | if vizinho à direita de  $v != -1$  then
    |   |   |   | c1 = vizinho à direita de v;
    |   |   | end
    |   |   | if vizinho à esquerda de  $v != -1$  then
    |   |   |   | c2 = vizinho à esquerda de v;
    |   |   | end
    |   | end
    | end
    | qnt_c1 = quantidade de focos que c1 contribuirá;
    | qnt_c2 = quantidade de focos que c2 contribuirá;
    | c = melhor candidato dentre c1 e c2;
    | S = S  $\cup$  c;
    | for cada foco  $f$  de  $c$  do
    |   | focos[f] = 1;
    | end
end
ordena S;
end

```

Algoritmo 3: Algoritmo Gulosso

Para a analisar a complexidade espacial, basta somar o custo $O(N+E+N*2*F)$ do grafo G (global), mais o custo da lista de solução final ($O(N)$) (global) e somar o custo das $2N^2 + 1$ chamadas recursivas, onde cada uma delas irá possuir seu vetor de solução parcial. Tal análise seja a conclusão que gasta-se $O(N+M+N*2*F+N+(2N^2+1)+N)$, como $N = M$ para grafos em anel, temos $O(2N^3 + 5N + 2NF)$. Para grandes quantidade de vértices, a ordem de complexidade espacial também será dada por $O(N^3)$.

Pode ser visto que tanto a ordem de complexidade temporal quanto a espacial é dada polinomialmente em função de N , M e por F (onde N = número de vértices, M = número de arestas e F = número de focos, R).

2.2. Programação Dinâmica

Dado N vértices, será necessário $O(2NF)$ operações para criar a Matriz inicial, N operações para criar a tabela de correspondência entre o elemento da linha e o vértice no grafo.

O pior caso desse algoritmo é quando cada vértice do grafo possui um foco em que é único. Com isso, será necessário avaliar todas as configurações, ou seja, adicionar um vértice em cada iteração para cada uma das N possíveis subsoluções, e são no máximo N^2 subsoluções possíveis. Para cada uma dessas N^2 subsoluções são feitas algumas operações constantes (por exemplo a de atualização dos índices) e operações que dependem da quantidade F de focos do grafo G para calculo nos índices na matriz, analisando a ordem assintótica será $O(2NF + N^2F)$, e para valores grandes quantidades de vértices será dado por $O(N^3)$.

Já o custo espacial é menor, pois a abstração do grafo (lista de adjacência e a lista de focos) ocupa $O(N+M+F)$, a matriz para o calculo ocupa $O(2NF)$ e o vetor solução $O(N)$. No total, o custo espacial assintótico do algoritmo de programação dinâmica é dado por $O(N^2)$.

2.3. Algoritmo Guloso

É claro perceber que o algoritmo guloso desenvolvido possui ordem de complexidade polinomial, tanto em análise temporal quanto a espacial. Para a criação do vetor G_ring , é necessário N operações, dado ao fato que a partir de um vértice, sempre irá apenas para um vértice diferente e nunca retornará para um vértice anterior. Já para a inicialização do vetor $focos$ de tamanho F , é necessário F operações para inicializar as posições.

Para encontrar o primeiro vetor para a solução, gasta-se no pior caso $O(2VF)$, pois para cada vértice do grafo, faz necessariamente duas atualizações, e a cada vértice adicionado, verifica se existem alguma posição do vetor de focos tal que ainda é zero.

Após isso, gasta-se $O(F)$ para atualizar o vetor de focos, que representará os focos cobertos pela solução parcial.

A cada laço do *while* de adição de novos vértices, gasta-se $O(F)$ para verificar se a solução atual não é satisfazível ainda, gasta-se também $O(V)$ para copiar e atualizar a lista G_ring_tmp , para definir os dois candidatos gasta-se $O(2V)$ comparações. Por fim, para avaliar qual dos vértices deve ser adicionado naquele momento, gasta-se um tempo constante $O(1)$, pois sempre cada vértice terá dois focos e como usa-se um vetor, o acesso a essas informações necessárias são feitas em $O(1)$.

Para o pior caso, ou seja, quando cada vértice possui um foco que só ele pode contribuir para a solução, o laço *while* irá ser executado $N-1$ vezes. Ao final final de todo o processo, o algoritmo deverá ordenar a solução em ordem crescente, para o pior caso, gasta-se $O(NlogN)$.

Com tudo, fica claro que a ordem de complexidade temporal total do algoritmo é $O(F + 2NF + (N-1)*(F+N+2N + 1) + NlogN = O(NF + N^2 + NlogN)$, para uma grande quantidade de vértices, $O(N^2)$.

Já para a ordem de complexidade espacial, pode ser visto que além de algumas variáveis de auxilio, o algoritmo precisa da representação do grafo, implementado em uma lista de adjacência $O(N+M)$ (como o grafo é em anel $N = M$) , mais os focos de cada vértice (como abstração do grafo) $O(N)$, dois vetores para representar o grafo na forma de lista $O(N)$, e um vetor de focos $O(F)$. Com isso, têm-se que a complexidade espacial do algoritmo é dada por $O(N+F)$, para uma grande quantidade de vértices em relação a quantidade de focos, vemos que o custo espacial do programa dependerá linearmente em função do número de vértices.

Pode ser visto então que tanto a ordem de complexidade temporal quanto a espacial é dada polinomialmente em função de N , M e F (onde N = número de vértices, M = número de arestas e F = número de focos, R). Além disso, a ordem de complexidade temporal é menor que o dos algoritmos anteriores, isto é, o de programação dinâmica e o busca por força bruta. No entanto, a solução gulosa se trata de um algoritmo heurístico, onde não garante a otimalidade da solução, apenas garante encontrar uma solução possível.

3. Exercício 3 – Implemente os algoritmos propostos no Exercício 1 na linguagem de programação C, C++, Java ou Python

Código fonte implementado em linguagem Python no apêndice deste relatório e também junto a pasta do projeto.

4. Exercício 4 – Execute testes da implementação do Exercício 3, propondo instâncias interessantes para o Problema ZikaZeroAnelDual. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n , m e r) comparando os três algoritmos implementados.

Os testes a seguir foram feitos em um computador Desktop com processador *i7-3537* de 2,00Ghz e com 8GB de memória RAM. Foram abordados dois casos , o primeiro onde a solução ótima é composta por apenas dois vértices e o segundo caso é onde a solução de um grafo com N vértices e N focos distintos é composta por $N-1$ vértices.

Esses dois casos são interessantes pois avaliam a capacidade do algoritmo de achar rapidamente uma solução para o caso onde a solução é composta por poucos elementos e no segundo caso, o desempenho do algoritmo quando a entrada de dados é o pior caso, isto é, quando a solução envolve todos os elementos. Para cada caso foi criado instâncias variando a quantidade de vértice de 10 à 240 e por consequência a quantidade de focos.

A Figura 4 representa o teste para o primeiro caso abortado no experimento. A

linha vermelha representa o tempo de execução para o algoritmo de força-bruta, a linha preta para o algoritmo de programação dinâmica e o azul para o guloso.

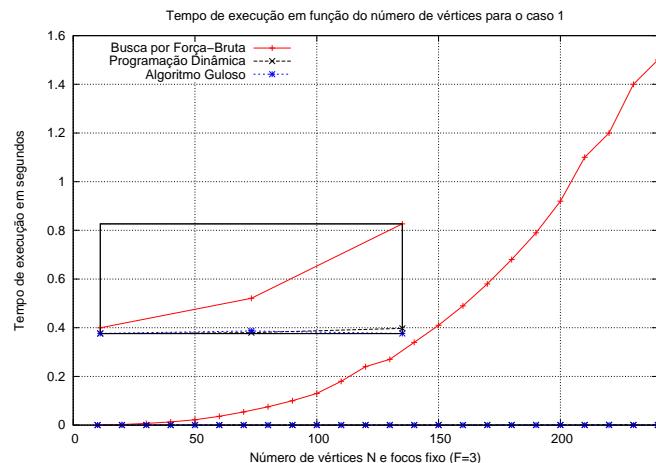


Figure 4. Tempo de execução em função tamanho do grafo para uma situação do caso 1.

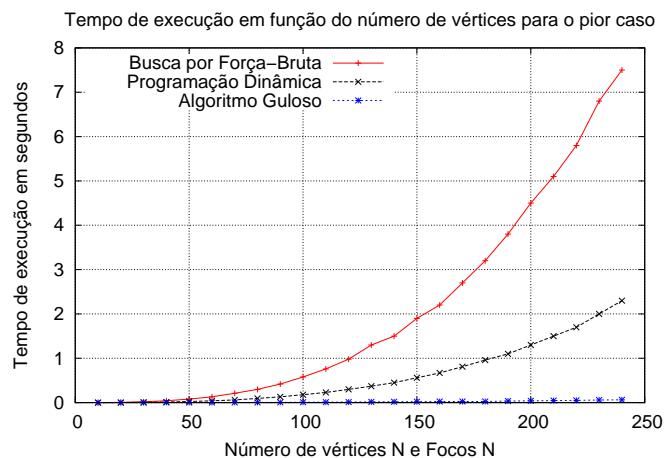
Pode ser visto que tanto a programação dinâmica e o guloso encontram uma solução rapidamente, o algoritmo de força bruta leva consideravelmente mais tempo, um dos motivos ele é recursivo o outro é porque ele utiliza pouca informação sobre a configuração do grafo, logo precisa percorrer mais possibilidades de configuração, por exemplo, ele avalia os caminhos $\{v_1, v_2, v_3\}$ e $\{v_3, v_2, v_1\}$ embora sejam iguais para o problema. Mais ainda, percebe-se que o tempo de execução para o guloso e dinâmico cresce muito pouco em função número de vértices para uma pequena quantidade de focos (no teste, $F = 3$).

A Figura 4 representa o teste para o segundo caso abortado no experimento. Pode-se ver que, para uma quantidade de vértices e de focos maiores ou igual à 50, a diferença de tempo entre eles já eram perceptíveis, o algoritmo de força-bruta precisou de mais tempo de processamento que os outros dois, o algoritmo de programação dinâmica foi o segundo mais rápido e por fim, o algoritmo guloso foi o mais rápido consideravelmente em relação aos demais.

Para ambos os casos de teste acima, os algoritmos encontram a solução correta. Embora que para o guloso, não há garantia da otimalidade, para o primeiro caso, como existia apenas uma solução possível para cada grafo, ele sempre a encontrou. Já para o segundo caso, como a solução era quase todos os focos, a solução encontrada também foi ótima para os testes realizados.

Também foi executada as instâncias de interessantes propostas pelo monitor da matéria de PAA. Em todos os casos ao algoritmo guloso foi mais rápido, em seguida o programação dinâmica. As soluções de força bruta e de dinâmico foram equivalentes as informadas no conjunto de testes, já para as soluções do guloso, como podia se esperar, houveram algumas diferenças.

Figure 5. Tempo de execução em função tamanho do grafo para uma situação de caso 2.

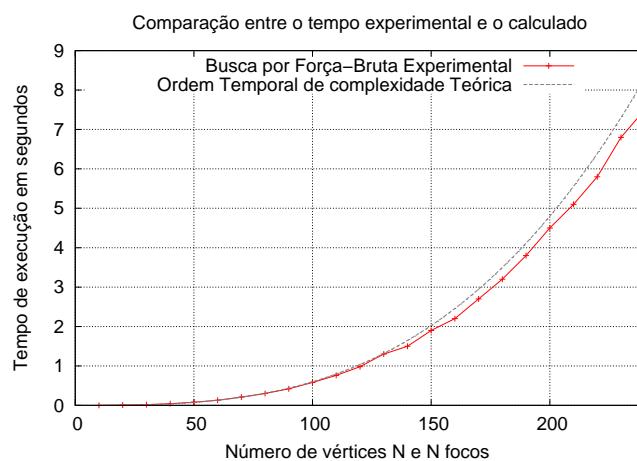


5. Exercício 5 – Compare a análise e a execução, respectivamente, Exercícios 2 e 4. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

Foi criado três gráficos, um para cada algoritmo, onde é mostrado o tempo de execução experimental monitorado para os algoritmos em instâncias do pior caso (i.e., quando cada vértice possui um foco importante para a solução) e o tempo de execução teórico (a partir da ordem de complexidade temporal calculada anteriormente e considerando a execução em um computador de 2GHz).

A Figura 6 é a comparação do tempo de execução para o algoritmo de força-bruta. A partir desse gráfico, pode-se perceber que o tempo de execução teórico e o experimental tem uma forte semelhança. Com isso, percebe-se que realmente a ordem de complexidade temporal calculada corresponde ao algoritmo implementado.

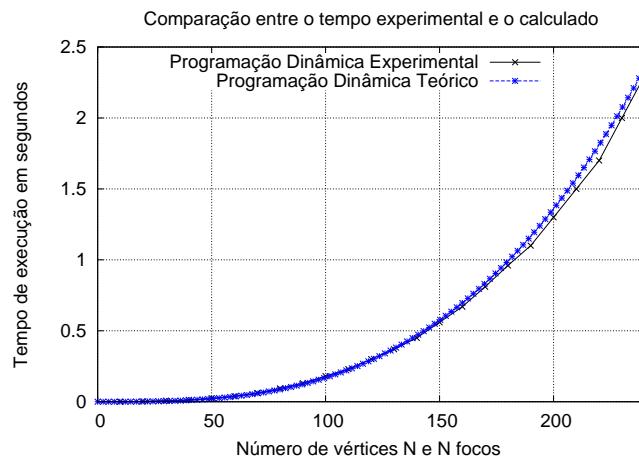
Figure 6. Comparação entre o tempo de execução do algoritmo de Força-Bruta teórico e experimental



A Figura 7 é a comparação do tempo de execução para o algoritmo de

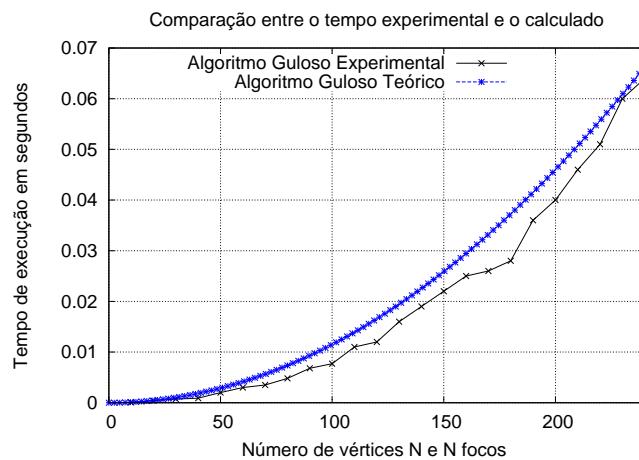
programação dinâmica. Assim como o anterior, pode se ver que o ambos os tempos (teórico e a experimental) também foram semelhantes. A ordem dos dois é a mesma.

Figure 7. Comparação entre o tempo de execução do algoritmo Dinâmico teórico e experimental



A Figura 8 é a comparação do tempo de execução para o algoritmo guloso. Assim como os outros, o tempo teórico foi semelhante ao tempo experimentalmente, mais ainda, o tempo teórico foi um limite superior para o tempo experimental.

Figure 8. Comparação entre o tempo de execução do algoritmo Guloso teórico e experimental



Por fim, conclui-se que para os três algoritmos, a ordem assintótica de complexidade temporal teórica foi analisada corretamente, os testes experimentais realizados corroboram com esse resultado. Além disso, embora o algoritmo guloso não garanta a otimalidade da solução, ele encontra uma solução mais rapidamente que os demais.

A. Código Fonte

A seguir serão mostrados os três códigos fontes para cada um dos três algoritmos pedidos.

A.1. Força Bruta

```
# coding: utf-8
import time
import os
import sys

Grafo = []
focos = []

#-----

class lista_adjacencia(object):

    def __init__(self,N,M):
        self.N = N
        self.M = M
        self.R = 0
        self.lista_adjacencias = [ [] for x in xrange( N+1 ) ]
        self.focos = [ [] for x in xrange( N+1 ) ]

    def add_adjacente(self, v1,v2):
        self.lista_adjacencias[v1].append(v2)

    def get_adjacentes_of_v1(self,v1):
        return self.lista_adjacencias[v1]

    def getN(self):
        return self.N

    def getM(self):
        return self.M

    def setR(self,R):
        self.R = R

    def getR(self):
        return self.R

    def add_focos(self,v1,foco1,foco2):
        self.focos[v1].append(foco1)
        self.focos[v1].append(foco2)

    def get_focos_of_v1(self,v1):
        return self.focos[v1]

# -----
```

```

class Solutions(object):
    def __init__(self):
        self.solutions = []
        self.found = False

    def add_partial_only_better(self,partial):
        if(len(self.solutions)!=0):
            if( len(partial)< len(self.solutions)):
                self.solutions = partial
            elif (len(partial) == len(self.solutions)):

                soma1 = sum(partial)
                soma2 = sum(self.solutions)

                if(soma1==soma2):
                    S1 = min(partial)
                    S2 = min(self.solutions)
                    if(S1 < S2):
                        self.solutions = partial

            else:
                self.solutions = partial

    def get_theonly_solution (self):
        self.solutions.sort()
        return self.solutions

# ----

def reader_file_PAA(name_file, Grafo):
    count = 0
    M = 0
    R = 0
    N = 0
    M = 0

    Grafo = []
    for line in open(name_file,"r").readlines():
        line = line.replace("\t", " ")
        line = line.replace("\n", "")
        v_temp = line.split(" ")
        if(count == 0):
            N = int(v_temp[0])
            M = int(v_temp[1])
            Grafo = lista_adjacencia(N,M)

        elif (count <= M):
            Grafo.add_adjacente( int(v_temp[0]), int(v_temp[1]) )

```

```

        Grafo.add_adjacente( int(v_temp[1]), int(v_temp[0]) )

    elif (count == (M+1)):
        Grafo.setR(int(v_temp[0]))
        R = int(v_temp[0])
    else:
        Grafo.add_focos(count-M-1,int(v_temp[0]), int(v_temp[1]))
    count+=1

return Grafo

#-----
def valida (Grafo,solution):
    valido = True
    focos = [ [] for x in xrange( Grafo.getR() + 1 )]

    if len(solution) != 0:
        for x in solution:
            focos_v1 = Grafo.get_focos_of_v1(x)
            for y in focos_v1:
                focos[y] = 1

        for x in xrange( 1, Grafo.getR() + 1 ):
            if focos[x] != 1:
                valido = False
    else:
        valido = False

    return valido

def Proximos (Grafo, solution):
    v1 = solution[-1]
    adjs = Grafo.get_adjacentes_of_v1(v1)

    v2 = []
    v3 = []

    if (adjs[0] not in solution):
        v2 = adjs[0]

    if (adjs[1] not in solution):
        v3 = adjs[1]

    return v2, v3

def forca_bruta(Grafo, partial,i,sols):
    i+=1

```

```

if(valida(Grafo,partial)):
    sols.add_partial_only_better(partial)
    return
else:
    if(len(partial) == 0):
        for x in xrange(1, Grafo.getN() ):
            partial = [x]
            forca_bruta(Grafo, partial,i,sols)
    else:
        v1,v2 = Proximos (Grafo,partial)
        partial1 = partial[:]
        partial2 = partial[:]
        if(v1 != []):
            partial1.append(v1)
            forca_bruta(Grafo, partial1,i,sols)

        if(v2 != []):
            partial2.append(v2)
            forca_bruta(Grafo, partial2,i,sols)

#-----#
if __name__ == '__main__':
    name_file = sys.argv[1]
    Grafo = []
    Grafo = reader_file_PAA(name_file, Grafo)
    sols = Solutions()
    forca_bruta(Grafo, [],1,sols)
    result = sols.get_theonly_solution()
    out_file = open(sys.argv[2],"w")
    msg = "%s" % result
    out_file.write( msg.replace("[","").replace("]", "")
                    .replace(","," ") )

```

A.2. Programação Dinâmica

```

# coding: utf-8
import math
import os
import sys

Grafo = []

#-----#
class lista_adjacencia(object):

    def __init__(self,N,M):
        self.N = N

```

```

        self.M = M
        self.R = 0
        self.lista_adjacencias =[] for x in xrange( N+1 )
        self.focos = [] for x in xrange( N+1 )

def add_adjacente(self, v1,v2):
    self.lista_adjacencias[v1].append(v2)

def get_adjacentes_of_v1(self,v1):
    return self.lista_adjacencias[v1]

def getN(self):
    return self.N

def getM(self):
    return self.M

def setR(self,R):
    self.R = R

def getR(self):
    return self.R

def add_focos(self,v1,foco1,foco2):
    self.focos[v1].append(foco1)
    self.focos[v1].append(foco2)

def get_focos_of_v1(self,v1):
    return self.focos[v1]

def getSolution(self,sols,table):
    subgrafo = []
    for y in xrange(len(sols)):
        v_init = sols[y][0]
        size_sol = sols[y][1]
        subsolution = []

        while ((size_sol)>=1):
            subsolution.append(table[v_init])
            size_sol-=1
            v_init+=1
            if(v_init==len(table)):
                v_init=0

            if(len(subgrafo)!=0):
                if( len(subsolution)< len(subgrafo)):
                    subgrafo = subsolution
                elif (len(subsolution) == len(subgrafo)):
```

```

        soma1 = sum(subsolution)
        soma2 = sum(subgrafo)
        if(soma1==soma2):
            S1 = min(subgrafo)
            S2 = min(subsolution)
            if(S1 > S2):
                subgrafo = subsolution
        else:
            subgrafo = subsolution

    subgrafo.sort()

    return subgrafo

# -----
def reader_file_PAA(name_file, Grafo):
    count = 0
    M = 0
    R = 0
    N = 0
    M = 0

    Grafo = []
    for line in open(name_file, "r").readlines():
        line = line.replace("\t", " ")
        line = line.replace("\n", "")
        v_temp = line.split(" ")

        if(count == 0):
            N = int(v_temp[0])
            M = int(v_temp[1])
            Grafo = lista_adjacencia(N,M)

        elif (count <= M):
            Grafo.add_adjacente( int(v_temp[0]), int(v_temp[1]) )
            Grafo.add_adjacente( int(v_temp[1]), int(v_temp[0]) )

        elif (count == (M+1)):
            Grafo.setR(int(v_temp[0]))
            R = int(v_temp[0])

        else:
            Grafo.add_focos(count-M-1,int(v_temp[0]), int(v_temp[1]))

        count+=1

    return Grafo

```

```

#-----

def construct_M (Grafo, Matrix):
    i = 0
    v = 1
    v_ant = 0
    N = Grafo.getN()
    table=[0 for x in range(0,N)]
    while (i < N):
        table[i]=v
        focos_v1 = Grafo.get_focos_of_v1(v)
        for y in focos_v1:
            Matrix[i][y-1] = 1
            Matrix[i+N][y-1] = 1

        adjs = Grafo.get_adjacentes_of_v1(v)

        if (adjs[0] == v_ant):
            v_ant = v
            v = adjs[1]

        else:
            v_ant = v
            v = adjs[0]

        i+=1

    return Matrix, table

def dinamico(Grafo):
    N = Grafo.getN()
    F = Grafo.getR()
    H = 2*N
    v_sol = 0
    size_sol =0
    solution = []

    Matrix = [[0 for x in range(F)] for y in range(H)]
    Matrix,table = construct_M (Grafo, Matrix)

    achou = False
    carry = 1
    cnt_carry = 0
    caminho = 2
    for i in xrange(N, N*N):

        for j in xrange(0,F):
            Matrix [cnt_carry+ N ][j] = Matrix [cnt_carry+ N ][j]
            or Matrix [ (carry + i) \%N ][j]

```

```

cnt_carry+=1

if (cnt_carry == N):
    cnt_carry =0
    carry+=1
    if (achou):
        break

if (0 not in Matrix[cnt_carry+ N ]):

    v_sol = cnt_carry
    size_sol = carry
    solution.append([v_sol,size_sol])
    achou = True

return solution,table

```

```

#-----
if __name__ == '__main__':
    name_file = sys.argv[1]
    Grafo = []
    Grafo = reader_file_PAA(name_file, Grafo)

    sols,table = dinamico(Grafo)
    solution = Grafo.getSolution(sols,table)

    out_file = open(sys.argv[2],"w")
    msg = "%s" % solution
    out_file.write( msg.replace("[","").replace("]", "") .
                    replace(", ", " ") )

```

A.3. Guloso

```

# coding: utf-8
import math
import os
import sys

Grafo = []

#-----

class lista_adjacencia(object):

```

```

def __init__(self,N,M):
    self.N = N
    self.M = M
    self.R = 0
    self.lista_adjacencias = [ [] for x in xrange( N+1 ) ]
    self.focos = [ [] for x in xrange( N+1 ) ]

def add_adjacente(self, v1,v2):
    self.lista_adjacencias[v1].append(v2)

def get_adjacentes_of_v1(self,v1):
    return self.lista_adjacencias[v1]

def getN(self):
    return self.N

def getM(self):
    return self.M

def setR(self,R):
    self.R = R

def getR(self):
    return self.R

def add_focos(self,v1,foco1,foco2):
    self.focos[v1].append(foco1)
    self.focos[v1].append(foco2)

def get_focos_of_v1(self,v1):
    return self.focos[v1]

# ----

def reader_file_PAA(name_file, Grafo):
    count = 0
    M = 0
    R = 0
    N = 0
    M = 0

    Grafo = []
    for line in open(name_file,"r").readlines():
        line = line.replace("\t", " ")
        line = line.replace("\n", "")
        v_temp = line.split(" ")

        if(count == 0):

```

```

        N = int(v_temp[0])
        M = int(v_temp[1])
        Grafo = lista_adjacencia(N,M)
    elif (count <= M):
        Grafo.add_adjacente( int(v_temp[0]), int(v_temp[1]) )
        Grafo.add_adjacente( int(v_temp[1]), int(v_temp[0]) )
    elif (count == (M+1)):
        Grafo.setR(int(v_temp[0]))
        R = int(v_temp[0])
    else:
        Grafo.add_focos(count-M-1,int(v_temp[0]), int(v_temp[1]))

    count+=1

    return Grafo

#-----



def construct_ring (Grafo):
    v = 1
    v_ant = 0
    G_ring = []
    while (len(G_ring) < Grafo.getN()):
        adjs = Grafo.get_adjacentes_of_v1(v)
        if (adjs[0] == v_ant):
            v_ant = v
            v = adjs[1]
            G_ring.append(v)
        else:
            v_ant = v
            v = adjs[0]
            G_ring.append(v)

    return G_ring


def get_first(Grafo,G_ring):
    v_first = 0
    focos = [0 for y in range(Grafo.getR()) ]
    for v in G_ring:
        focos_v1 = Grafo.get_focos_of_v1(v)
        for f in focos_v1:
            focos[f-1]=1
    if (0 not in focos ):
        v_first = v
        break
    return v_first


def increment(solution, G_ring,Grafo,focos_faltantes):

```

```

for i in xrange (0, len(G_ring)):
    if (G_ring[i] in solution):
        G_ring[i]=-1

c1 = -1
c2 = -1

for i in xrange (0, len(G_ring)):
    if(G_ring[i] == -1):
        if( i == len(G_ring)-1):
            ii = 0
            if(G_ring[ii] is not -1):
                c2 = G_ring[ii]
        else:
            if(G_ring[i+1] is not -1):
                c2 = G_ring[i+1]

    if(G_ring[i] == -1):
        if( i-1 != -1):
            if(G_ring[i-1] != -1):
                c1 = G_ring[i-1]
        else:
            ii = len(G_ring)-1
            if(G_ring[ii] != -1):
                c1 = G_ring[ii]

qnt_c1 = 0
qnt_c2 = 0

if(c1 !=-1):
    focos_c1 = Grafo.get_focos_of_v1(c1)
    for f in focos_c1:
        if(focos_faltantes[f-1] == 0):
            qnt_c1+=1
if(c2 !=-1):
    focos_c2 = Grafo.get_focos_of_v1(c2)
    for f in focos_c2:
        if(focos_faltantes[f-1] == 0):
            qnt_c2+=1
if(qnt_c1>=qnt_c2):
    solution.append(c1)
    for f in focos_c1:
        focos_faltantes[f-1] =1
else:
    solution.append(c2)
    for f in focos_c2:
        focos_faltantes[f-1] =1

return solution,focos_faltantes

```

```

def guloso(Grafo):
    N = Grafo.getN()
    F = Grafo.getR()

    v_sol = 0
    size_sol = 0
    solution = []
    focos_faltantes = [0 for y in range(Grafo.getR())]
    G_ring = construct_ring(Grafo)
    solution.append(get_first(Grafo, G_ring))
    focos_v1 = Grafo.get_focos_of_v1(solution[0])
    focos_faltantes[focos_v1[0]-1] = 1
    focos_faltantes[focos_v1[1]-1] = 1

    while (0 in focos_faltantes):
        solution, focos_faltantes = increment(solution, G_ring, Grafo,
                                                focos_faltantes)

    solution.sort()
    return solution

```

```

if __name__ == '__main__':
    name_file = sys.argv[1]
    Grafo = []
    Grafo = reader_file_PAA(name_file, Grafo)

    start = time.clock()
    sol = guloso(Grafo)
    end = time.clock()
    out_file = open(sys.argv[2], "w")
    msg = "%s" % sol
    out_file.write( msg.replace("[", ""))
                    .replace("]", "").replace(", ", " ") )

```

Projeto e Análise de Algoritmos - Paradigmas: Problema ZikaZeroAnelDual

Edson B. de Lima¹

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

1. Descrição do problema

O vírus da zika causa a doença também conhecida como zika. A transmissão do vírus se dá através da picada do mosquito Aedes aegypti. Para prevenir danos à saúde de todos, é necessário combater a propagação do mosquito. Cerca de 80% dos criadouros estão localizados dentro das residências ou nos locais de trabalho, portanto, em espaços privados. A dificuldade está em conseguir voluntários que tenham acesso a todos esses locais. Considere uma Campanha Corrente do Bem em que as pessoas desafiam-se a formar linhas de frente para acabar com os criadouros. Inicialmente, há um primeiro voluntário, que indica um amigo, que por sua vez indica outro, e assim sucessivamente, até que o último indica o primeiro (fechando um ciclo). Além disso, nessa campanha, cada voluntário deve visitar a exatamente dois criadouros. Nesse contexto, existe o interesse no seguinte problema.

Problema ZikaZeroAnelDual. Dados um grafo anel (aquele em que todo vértice conecta-se a exatamente outros dois) $G(V, A)$, em que V é o conjunto de n voluntários e A é o conjunto de $m = |V|$ laços de amizade, 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, sendo que $|R(v)| = 2, \forall v \in V$. **O objetivo é selecionar o menor número de voluntários $V' \subset V$, tal que, todo foco é acessado, por pelo menos um voluntário $v \in V'$, e o grafo induzido por V' em G é conexo.**

2. Modelagem

Dada a instância que representa o problema ZikaZeroAnelDual, o objetivo é encontrar entre o espaço de soluções possíveis, uma configuração que seja ótima, isto é, o menor conjunto de vértices no grafo G' induzido em G tal que G contenha todos os focos e seja conexo.

2.1. Força-bruta

Seja $G = (V, A)$ o grafo que representa a instância do problema ZikaZeroAnelDual, tem-se que as possibilidades que cada vértice $v \in V$ pode obter ao realizar um percurso em G consiste na recorrência dos percursos:

$$P_1(n) = \{\emptyset, n = 0, v- > P(\text{adj}(v^+))\} \quad (1)$$

e

$$P_2(n) = \{\emptyset, n = 0, v- > P(\text{adj}(v^-))\} \quad (2)$$

Onde $\text{adj}(v^+)$ representa o vértice adjacente à direita de v no percurso em sentido horário (1) e $\text{adj}(v^-)$ o vértice adjacente à esquerda de v no percurso em sentido anti-horário (2). Por exemplo, partindo do vértice v_i há a possibilidade de encontrar $n - 1$ percursos tanto à esquerda quanto à direita.

Assim, tem-se que para cada vértice $v \in V$, o número de possibilidades em ambos os sentidos é igual a $2(n - 1)$. Como são n vértices no grafo G , o número total de possibilidades corresponde a $2n(n - 1)$.

O processo de modelagem deve utilizar uma estrutura de dados em lista para armazenar todas as configurações de percurso no qual os vértices presentes em um percurso j contenham todos os focos, $1 \leq j \leq 2n(n - 1)$. Dessa forma, a lista deve ter no pior caso tamanho $2n(n - 1)$ configurações. Sendo preciso realizar uma busca linear sobre a lista para encontrar o menor valor. O custo total desse processo deve ser

$$T(n) = 2(2n(n - 1)) = \mathcal{O}(n^2) \quad (3)$$

2.2. Programação Dinâmica

A abordagem do algoritmo em Programação Dinâmica segue a estratégia de memorização [Cormen] [Kleinberg], reaproveitando a solução de subproblemas, considerando um subcaminho já seguido no grafo. Ou seja, o processo de busca do vértice i ao vértice j deve seguir a recorrência:

$$\text{OPT}(G, i, j) = \{f(j, j - 1) + \text{OPT}(G, i, j - 1)\}, \quad (4)$$

onde $f(j, j - 1)$ indica a função que marca os focos na aresta $(j, j - 1)$. A cada iteração, o algoritmo verifica se esse valor foi calculado. Caso isso aconteça, o algoritmo reaproveita o valor, consultando-o na memória sem marcar os vértices. Por outro lado, se o valor ainda não foi calculado, o algoritmo realiza o processo de visita nos vértices, marcando os focos correspondentes.

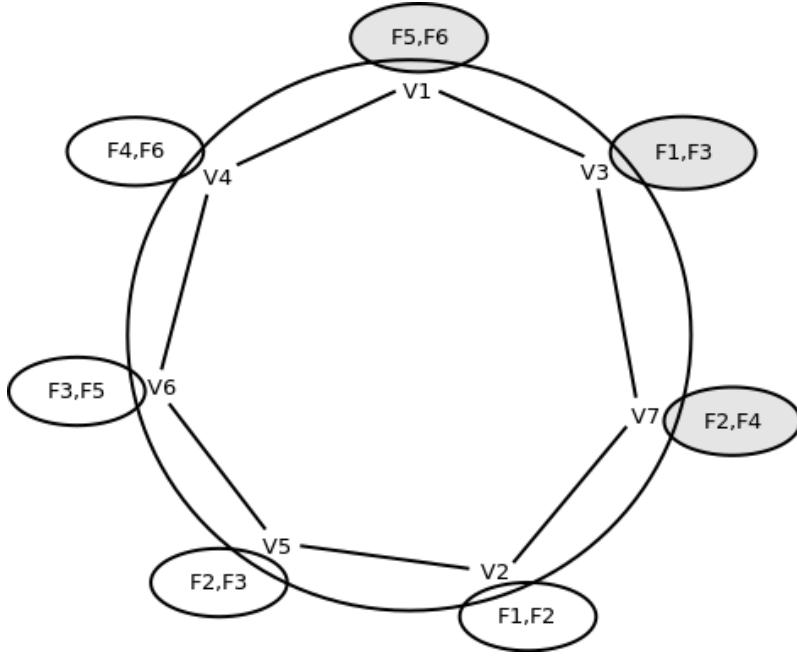


Figura 1. Grafo que representa uma instância do problema ZikaZeroAnelDual

A Figura (1) representa um grafo G como instância do problema ZikaZeroAnel-Dual com $|V| = |E| = 7$ e $|F| = 6$ na qual cada vértice contém exatamente dois focos. Na abordagem dinâmica, por exemplo ao realizar uma busca em G do vértice v_1 ao v_4 há sobreposição de buscas a partir dos vértices $\{v_3, v_7, v_2, v_5, v_6\}$ ao vértice v_4 . Nesse caso para cada v_{i-k} , ou seja, aqueles vértices mais à esquerda de v_i , tem-se $n - k$ sobreposições no processo de busca.

2.3. Algoritmo guloso

O método guloso consiste em obter o menor conjunto de vértices com todos os focos através de uma estratégia aproximada [Vazirani 2001] considerando um peso nas arestas que contém vértices adjacentes com maior número de focos. O algoritmo deve, então, ordenar as arestas de acordo com seu peso, de forma que as primeiras arestas representam aquelas com maior número de focos nos seus vértices.

Tal estratégia não garante a optimalidade no caso de o grafo conter arestas com o mesmo número de focos em ambos os vértices. No entanto, a solução gulosa garante uma boa taxa de aproximação da solução ótima, por encontrar o menor conjunto de vértices com mais focos seguindo a abordagem *stays ahead* [Kleinberg].

3. Complexidade temporal e espacial

Considerando a análise do problema nas abordagens por força-bruta, programação dinâmica e guloso, observa-se menor complexidade temporal e espacial nas duas últimas abordagens em comparação com o método da força bruta.

3.1. Complexidade Força Bruta

Considerando a análise de complexidade da equação de recorrência (1), o método de força bruta deve realizar o processo de busca para encontrar todos os focos a partir de cada vértice do grafo $G = (V, E)$, o que deve custar.

$$T(n) = O(|V|^3|F|) \quad (5)$$

Pois, como a busca em profundidade custa $O(|V| + |E|)$ [Cormen] [Kleinberg], o processo de busca para todos os vértices de G deve custar $O(|V|(|V| + |E|)) = O(|V|^3)$ no pior caso.

3.2. Complexidade Programação Dinâmica

Na complexidade espacial, se a solução para o subcaminho $p = (u, v)$ foi calculada, então o subcaminho (v, u) utiliza o valor previamente calculado por (u, v) . Dessa forma, o número de operações realizadas em memória corresponde ao número de arestas gravadas nos subproblemas de acordo com a recursão (4) que representa o número de arestas no anel, isto é:

$$T(n) = O(n) \quad (6)$$

Pois o número de arestas $|E| = |V| = n$, sendo a complexidade espacial linear em função do número de arestas. Já a complexidade temporal deve levar em consideração o número de chamadas recursivas para cada um dos $(n - k)$ vértices sobrepostos, como mencionado anteriormente. Assim, para cada vértice v_i tem-se $n - 1$ chamadas recursivas, o que custa no total $O(n^2)$ operações, acessando os cálculos na memória.

3.3. Complexidade Algoritmo Gulosso

As etapas do algoritmo gulosso consistem em:

- a) Ordenar o conjunto de arestas de acordo com a maior quantidade de focos distintos e a menor soma
- b) Percorrer as arestas em sequencia ordenada, marcando os vértices correspondentes até que todos os focos fiquem marcados.

O procedimento do item a) tem complexidade $T(n) = O(m\log(m))$ determinado pela ordenação [Cormen], já o item b) custa no pior caso $O(n)$ operações para percorrer o arranjo de arestas ordenadas, sendo $m = |E|$.

4. Implementação

Para a implementação do problema foi utilizada a linguagem de programação Java 8 junto com a ferramenta Excel na análise dos resultados e relatórios. Em cada um dos três paradigmas a estrutura utilizada para o problema foi um grafo não-direcionado com focos nos vértices. Para cada aresta (u, v) no grafo há uma aresta (v, u) considerando a construção do grafo por lista de adjacências.

Foram utilizadas as bibliotecas auxiliares para leitura e gravação de dados, assim como operações em estrutura de dados auxiliades. São elas:

- a) **Arquivo.java**: utilizada para a manipulação dos dados, como abertura de arquivo, entrada de dados, e gravação dos resultadas na saída do arquivo
- b) **HashSet**: estrutura de dados utilizada para remover as duplicações em um vetor de dados e ordenar por índices

c) **ArrayList.java**: estrutura de dados utilizada para armazenar e manipular dados de vértices, arestas, focos, etc.

4.1. Implementação em Força Bruta

No paradigma de força bruta, o processo de marcação dos focos segue uma busca em profundidade (Depth First Search) na qual cada vértice vai sendo atingido a partir de um vértice inicial. Uma lista de valores booleanos é utilizada para marcar os focos que já foram atingidos passando-se por cada vértice. Dessa forma, o processo de busca termina quando dado um vértice inicial todos os focos são atingidos. Seguindo o proposto pela Equação (1), o processo DFS é executado a partir de cada vértice, sendo então preciso marcar os focos descobertos a cada iteração por vértice.

4.2. Implementação em Programação Dinâmica

O método por programação dinâmica reaproveita a estrutura de dados em grafo para a construção de um grafo não-direcionado $G = (V, E)$. Contudo, o algoritmo utiliza como memória extra [Erickson 2015] [Kleinberg] uma tabela de tamanho $mx f$, onde m representa o número de arestas $|E|$ e f o número de focos presente no grafo G .

A cada iteração o algoritmo verifica se alguma linha dessa tabela foi totalmente marcada, ou seja, com todos os focos atingidos. Enquanto os focos não foram todos descobertos, o algoritmo segue continuando o procedimento de realizar mesclagem de cada linha com sua linha sucessiva que representa a aresta adjacente no grafo. Tal processo é realizado no máximo $|E| - 1$ vezes, caso em que cada aresta foi mesclada com as demais no ciclo. Ao final das iterações, o algoritmo deve retornar o número da linha em que ocorreu *match*, isto é, todos os focos foram marcados.

Além da tabela que armazena as informações dos focos encontrados, o algoritmo também utiliza um vetor de inteiros para controlar a contagem do número de focos encontrados considerando cada linha da tabela. Quando o valor de algum item desse vetor atingir o total de focos, o algoritmo tem encontrado dos os focos em uma linha. Isso significa que o algoritmo encontrou uma aresta ou uma sequência destas no qual todos os focos foram atingidos.

5. Testes e experimentação

Os testes foram analizados com as seguintes entradas e medidos respectivamente os valores dos tempos para encontrar a solução nos métodos de Força Bruta e Programação Dinâmica.

Tabela 1. Teste e comparação do tempo de execução com método da Força Bruta

Análise de Execução Força Bruta			
Vértices	Arestas	Focos	Tempo (ms)
7	7	6	105
5	5	6	66

Os testes indicaram que em geral os tempos de execução utilizando a abordagem dinâmica foi melhor, como mostra a Tabela 2.

Tabela 2. Teste e comparação do tempo de execução com método de Programação Dinâmica

Análise de Execução Força Bruta			
Vértices	Arestas	Focos	Tempo (ms)
7	7	6	105
5	5	6	66

6. Comparação da análise e execução

Os testes e experimentação corroboram com a análise teórica de complexidade em cada método, visto que o método de Força Bruta se mostrou mais custoso em todas os casos testes enquanto que o método em PD indica mais eficiência considerando todas as entradas. Isso comprovado, pois enquanto o método em força bruta deve custar $O(|V|^2|F|)$ com a técnica de PD a complexidade é de $O(|V|)$.

Referências

Cormen, T. *Algoritmos Teoria e Prática*. 3th edition.

Erickson, J. (2015). *Algorithm Notes*.

Kleinberg, J. *Algorithm Design*. 3th edition.

Vazirani, V. (2001). *Approximation Algorithms*.



UNIVERSIDADE FEDERAL DE MINAS GERAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**PROJETO E ANÁLISE DE ALGORITMOS: TRABALHO
PRÁTICO ZIKAZEROANELDUAL**

MARIA LUIZA BURGARELLI ALVES DOS SANTOS

**BELO HORIZONTE
JUNHO 2016**

Resolução dos Exercícios

Exercício 1

Dado o problema do ZikaZeroAnelDual foram propostos diferentes algoritmos baseados em diferentes paradigmas, conforme solicitado na orientação do projeto.

O primeiro algoritmo a ser analisado é o algoritmo de força bruta. Este algoritmo é mostrado no Algoritmo 1. Inicialmente, são geradas listas de possibilidades partindo de todos os vértices, sem verificar se todos os focos estão sendo cobertos. As listas são geradas percorrendo-se o grafo em anel no sentido horário, primeiramente cobrindo uma volta inteira e incrementalmente retirando-se um elemento da lista para formar uma nova, até que sejam geradas listas de tamanho variável de $numero_{focos}/2$ até n .

Algoritmo 1: Algoritmo força bruta para descoberta do menor número de voluntários para cobertura dos focos em grafo em anel

Input: arquivo contendo:

$n \leftarrow$ número de voluntários

$m \leftarrow$ número de laços de amizade

$M_{amizade} \leftarrow$ matriz contendo os laços de amizade

$r \leftarrow$ número de focos a serem considerados

$M_{acessos} \leftarrow$ matriz de n linhas contendo os focos que cada voluntário tem acesso

Output: arquivo contendo os voluntários selecionados como solução

```

1 tamLista  $\leftarrow n$ 
2 for cada vértice do grafo em anel do
3   | Calcular lista de possibilidades de tamanho tamLista
   | tamLista  $\leftarrow tamLista - 1$ 
4 end
5 for cada lista de vértices do
6   | if lista cobre todos os focos then
7     |   | coverFocos  $\cup$  lista de vértices
8   | end
9 end
10 Ordenar o conjunto de listas em ordem crescente do número de vértices
11 Ordenar a primeira lista (menor) em ordem crescente dos vértices (índices)
12 solucao  $\leftarrow$  menor lista encontrada
13 Escrever solucao no arquivo de saída

```

Já o algoritmo guloso proposto é baseado no algoritmo guloso para cobertura de conjuntos conectada de Zhang et al. (2009). Este algoritmo pode ser visualizado no Algoritmo 2. É selecionado um vértice aleatoriamente para ser colocado no conjunto solução, uma vez que todos os conjuntos de cada vértice contém o mesmo número de elementos (2 focos). Em seguida, são selecionados todos os vértices adjacentes aos vértices que já estão no conjunto solução, bem como aqueles vértices cujos conjuntos

possuem pelo menos um foco em comum com os focos dos conjuntos dos vértices do conjunto solução. Para estes vértices selecionados é definido o menor caminho a partir dos vértices do conjunto solução, seja pelo sentido horário, ou pelo sentido anti-horário. São então calculadas as razões entre o número de vértices do menor caminho para cada vértice selecionado e o número de focos que são cobertos pelo caminho selecionado mas não estão no conjunto solução. Por fim, é selecionado o caminho com a menor razão e todos os focos pertencentes aos vértices deste caminho são adicionados na solução.

O último algoritmo proposto utiliza programação dinâmica e é mostrado no Algoritmo 3. Ele é baseado parcialmente no algoritmo apresentado nas discussões/orientações do projeto e no algoritmo descrito em alto nível em Ritt (2012). O algoritmo é baseado na recorrência (RITT, 2012):

$$S(C, i) = \begin{cases} 0, & \text{caso } i=0 \text{ e } C = \emptyset, \\ \infty, & \text{caso } i=0 \text{ e } C \neq \emptyset, \\ \min\{S(C \setminus C_i, i - 1), S(C, i - 1)\}, & \text{caso contrário.} \end{cases}$$

$S(C, i)$ é o tamanho da menor cobertura para C com conjuntos C_1, C_2, \dots, C_i .

Exercício 2

A seguir serão analisadas as complexidades temporal e espacial dos algoritmos propostos.

Análise de Complexidade Temporal

O Algoritmo 1 funciona com complexidade $O(2n^2)$ pois é necessário percorrer todas as possibilidades de todos os vértices correspondentes em um sentido, gerando $(n)(n - 3)$ listas. É necessário percorrer todas as listas para verificar se cobrem os focos, com complexidade $O(2n^2)$.

Já o algoritmo 2 funciona com complexidade $O(n \log n)$ pois não é necessário percorrer todos os elementos do grafo. A complexidade e a solução ótima obviamente dependerá da escolha inicial do vértice aleatório, mas em geral podemos dizer que possui esta complexidade.

Por fim, o algoritmo de programação dinâmica funciona com complexidade $O(n^2 + n)$, pois é necessário um loop de n^2 iterações, mais um loop sobre cada voluntário.

Pode-se verificar que o algoritmo dinâmico é ligeiramente melhor que o força bruta. No entanto, os melhores resultados foram obtidos pelo algoritmo guloso.

Algoritmo 2: Algoritmo guloso para descoberta do menor número de voluntários para cobertura dos focos em grafo em anel

Input: arquivo contendo:

- $n \leftarrow$ número de voluntários
- $m \leftarrow$ número de laços de amizade
- $M_{amizade} \leftarrow$ matriz contendo os laços de amizade
- $r \leftarrow$ número de focos a serem considerados
- $M_{acessos} \leftarrow$ matriz de n linhas contendo os focos que cada voluntário tem acesso

Output: arquivo contendo os voluntários selecionados como solução

- 1 $R \leftarrow$ conjunto solução (vértices)
- 2 $U \leftarrow$ conjunto dos focos cobertos pelos vértices solução $V \leftarrow$ coleção de todos os conjuntos de focos de cada vértice
- 3 $S \leftarrow$ conjunto de todos os vértices do grafo
- 4 Escolher um vértice S_i do grafo em anel aleatoriamente
- 5 Adicionar o vértice em R e os focos cobertos por ele em U
- 6 **while** $V \setminus U \neq \emptyset$ **do**
- 7 **for** cada vértice S_i pertencente a $S \setminus R$ **do**
- 8 **if** é adjacente a algum vértice de R ou possui um elemento em comum com algum vértice em R **then**
- 9 $caminhosHorarios \leftarrow$ caminho de R ao vértice S_i no sentido horário
- 10 $caminhosAnti \leftarrow$ caminho de R ao vértice S_i no sentido anti-horário
- 11 **end**
- 12 **end**
- 13 **for** cada vértice S_i pertencente a $S \setminus R$ **do**
- 14 Verificar se caminho horário ou anti-horário é o menor
- 15 **end**
- 16 **for** cada vértice S_i pertencente a $S \setminus R$ **do**
- 17 $numVert \leftarrow$ número de vértices do caminho
- 18 $cps \leftarrow$ número de elementos cobertos pelo menor caminho mas não cobertos por R .
- 19 Calcular razão $e \leftarrow numVert/cps$
- 20 **end**
- 21 $Ps \leftarrow$ caminho (vértices) com menor razão e $R \leftarrow R \cup Ps$ $U \leftarrow U \cup$ focos do caminho Ps
- 22 **end**
- 23 Ordenar R
- 24 Escrever R no arquivo de saída

Algoritmo 3: Algoritmo em programação dinâmica para descoberta do menor número de voluntários para cobertura dos focos em grafo em anel

Input: arquivo contendo:

$n \leftarrow$ número de voluntários

$m \leftarrow$ número de laços de amizade

$M_{amizade} \leftarrow$ matriz contendo os laços de amizade

$r \leftarrow$ número de focos a serem considerados

$M_{acessos} \leftarrow$ matriz de n linhas contendo os focos que cada voluntário tem acesso

Output: arquivo contendo os voluntários selecionados como solução

```

1  $S(0, \emptyset) \leftarrow 0$ 
2  $U \leftarrow 0$  for cada  $j$  de 1 até  $n^2$  do
3   |  $i \leftarrow 1$   $C \leftarrow 0$  while  $k > 0$  do
4     |   |  $b \leftarrow N \bmod 2$ 
5     |   | if  $b == 1$  then
6       |   |   |  $C \leftarrow C \cup voluntario_i$ 
7     |   | end
8     |   |  $k \leftarrow k/2$   $i \leftarrow i + 1$ 
9   | end
10  |  $U \leftarrow U \cup C$ 
11 end
12 for  $S$  pertencente a  $U$  do
13   |  $S(0, S) \leftarrow \infty$ 
14 end
15 for para cada voluntário  $A$  do
16   | for  $S$  pertencente a  $U$  do
17     |   | if  $S(i - 1, S) > S(i - 1, S \setminus A) + 1$  then
18       |   |   |  $S(i, S) \leftarrow S(i - 1, S \setminus A) + 1$ 
19     |   | end
20     |   | else
21       |   |   |  $S(i, S) \leftarrow S(i - 1, S)$ 
22     |   | end
23   | end
24 end
25 Ordenar  $S(A, F)$ 
26 Escrever  $S(A, F)$  no arquivo de saída

```

Análise de Complexidade Espacial

O Algoritmo 1 funciona com complexidade espacial $O(2n^2)$ pois é necessário gerar $(n)(n - 3)$ listas.

O algoritmo 2 funciona com complexidade espacial também de $O(n)$, pois são gerados 4 conjuntos de no máximo $2n$ elementos.

Por fim, o algoritmo dinâmico tem complexidade espacial de $O(n^2)$ também pois são gerados n conjuntos de tamanho n .

Exercício 3

O algoritmo proposto foi implementado em Java e foi enviado em anexo no portal *minha.ufmg*.

Exercício 5

Os resultados experimentais foram ligeiramente inferiores aos resultados esperados analiticamente, uma vez que foram utilizadas muitas estruturas de dados *ArrayList* e pela ordenação em sentido horário utilizada nas implementações, também utilizando a mesma estrutura de dados. No entanto, pelo desempenho geral, em comparação dos algoritmos, os resultados experimentais foram compatíveis com os analíticos.

Referências

RITT, M. **Algoritmos e Complexidade - Notas de Aula**. 2012. <http://www.inf.ufrrgs.br/~mrpritt/lib/exe/fetch.php?media=cmp155:notas-4227.pdf>. Citado na página 2.

ZHANG, Z.; GAO, X.; WU, W. Algorithms for connected set cover problem and fault-tolerant connected set cover problem. **Theoretical Computer Science**, Elsevier, v. 410, n. 8, p. 812–817, 2009. Citado na página 1.

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

Trabalho Prático: ZikaZeroAnelDual

Nícollas de Campos Silva

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

19 de junho de 2016

Resumo

Este documento tem como objetivo descrever a modelagem e as distintas abordagens de soluções propostas para o problema ZikaZeroAnelDual, bem como justificar as principais decisões e escolhas associadas a tais soluções. Além disso, discutimos o impacto de algumas instâncias do problema a fim de analisar a eficiência dos algoritmos propostos.

1 Distintas Abordagens

Nesta seção apresentamos três abordagens distintas para a solução do problema, baseado nas estratégias de: busca por força-bruta (ou busca exaustiva), programação dinâmica e algoritmo guloso. Para tal, primeiro apresentamos a modelagem dos dados, a fim de melhorar o entendimento das soluções propostas. Em seguida, apresentamos as soluções polinomiais para o problema, bem como os respectivos algoritmos.

1.1 Modelagem Proposta

A modelagem proposta consiste em um grafo $G = (V, E)$ não orientado no formato anel¹, onde os voluntários na ação contra o *zika-vírus* representam o conjunto V de vértices e os laços de amizades existente entre eles o conjunto E de arestas do grafo. Para tal abordagem, na representação do grafo optou-se por utilizar o conceito de lista de adjacências adaptada a dois vetores simples, visto que o formato em anel do grafo indica a existência de apenas dois vértices adjacentes. Neste intuito, utilizamos um vetor $D_{1 \times |V|}$ para marcar os vértices alcançáveis ao caminhar no sentido horário, e um vetor $E_{1 \times |V|}$ similar ao primeiro, porém com os vértices alcançáveis ao caminhar no grafo no sentido anti-horário.

Para representar os focos atingidos por cada um dos voluntários, utilizou-se apenas duas estruturas de dados visto que cada voluntário sempre terá acesso a apenas dois focos distintos entre si. Para tal, utiliza-se dois vetores, $F1$ e $F2$, de tamanhos $1 \times |V|$, que armazenam o índice do foco coberto. A escolha dessa modelagem dos focos se deve a praticidade e simplicidade de utilização, principalmente quando os grafos são razoavelmente pequenos. Por outro lado, ao utilizarmos apenas os vetores D e E para representar os próximos vértices, obtemos uma economia de memória significativa, quando comparada a uma lista ou matriz de adjacências, no escopo deste trabalho.

¹Um grafo em formato de anel é aquele em que todos os vértices se conectam exatamente a outros dois vértices.

A figura 1 abaixo demonstra a utilização de nossa modelagem para o grafo proposto como instância inicial para o problema *ZikaZeroAnelDual*. Nota-se que do grafo original resulta nos vetores D e E (b), que possuem valores correspondentes entre si, e nos vetores $F1$ e $F2$ (c) onde cada posição i destes vetores representam os focos atingidos pelos vértices V_i .

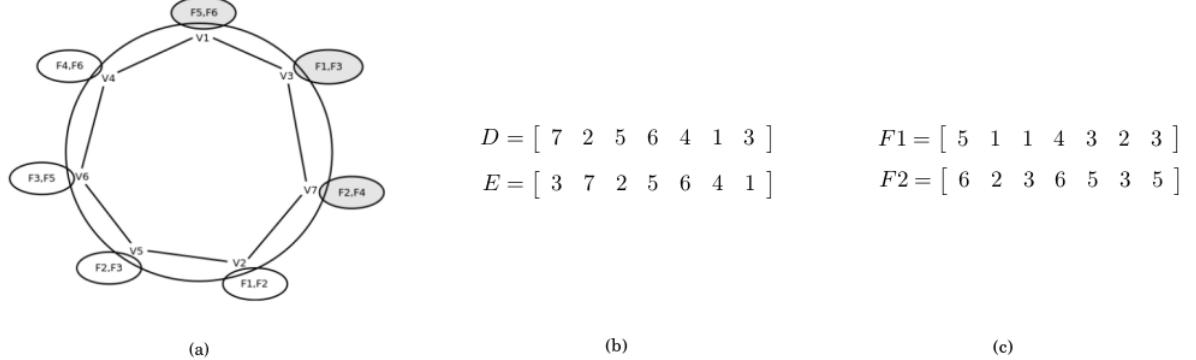


Figura 1: Aplicação da modelagem sobre a instância proposta na especificação deste trabalho. Em (a) temos o grafo proposto. (b) mostra os vetores D e E gerados. (c) mostra os vetores $F1$ e $F2$ que relaciona os voluntários aos focos.

Outro aspecto importante a ser destacado é o pré-processamento necessário para a leitura dos dados fornecidos em cada instância. Devido a escolha dos vetores D e E para representar a forma de caminhar pelo grafo, é necessário forçar que os vértices fornecidos pelo padrão de entrada estejam na ordem adequada. Neste intuito, o algoritmo 1 garante que um determinado vértice adicionado em um dos vetores (i.e., adicionados a direita), não seja novamente adicionado neste vetor (i.e., a direita) em outra iteração. Em outras palavras, após a leitura dos dados, os vetores D e E são analisados de forma a realizar uma troca sempre que houver mais de um vértice com o mesmo índice no mesmo vetor. Dessa forma, após o pré-processamento dos dados é garantido o êxito da modelagem proposta para quaisquer formato das instâncias. Previamente, pode-se notar que este algoritmo não é o responsável pela complexidade no limite assintótico de todas as demais tarefas a serem realizadas. Esta tarefa, possui complexidade temporal de $O(|V| \times |V - 1|) = O(|V|^2)$ e complexidade espacial de $O(|V| + |V|) = O(|V|)$, devido aos vetores D e E .

Algorithm 1 STEP-PROCESS(D, E)

```

1: for  $i = 1$  to  $|V|$  do
2:    $v \leftarrow D[i]$ 
3:   for  $j = i + 1$  to  $|V|$  do
4:     if  $v = D[j]$  then
5:       EXCHANGE( $D[j], E[j]$ )
6:     else
7:       if  $v = E[j]$  then
8:          $v \leftarrow E[j]$ 
9:       end if
10:      end if
11:    end for
12:  end for

```

1.2 Abordagem de Força Bruta

A estratégia por força bruta (ou busca exaustiva) é uma técnica com a característica de enumerar todos os possíveis candidatos da solução e checar cada candidato para saber se este satisfaz ou não ao enunciado do problema. De maneira geral, a estratégia de força bruta é capaz de gerar a solução ótima, caso esta seja viável, porém muitas das vezes com um custo computacional exponencial. Tal custo infere na principal desvantagem desta abordagem, visto que o número de soluções candidatas tende a ser grande, tornando a busca pela solução num processo inviável. Entretanto, o algoritmo implementado é uma estratégia força bruta com custo polinomial devido ao formato em anel do grafo.

Para o problema ZikaZeroAnelDual, a estratégia força bruta utilizada consiste em avaliar a partir de cada vértice do grafo, quantos próximos vértices são necessários para cobrir todos os focos. Basicamente, a partir do primeiro vértice, caminhamos no sentido horário do grafo sempre adicionando o vértice a direita na solução e verificando se os vértices selecionados cobrem todos os focos. Tal verificação se dá devido a função COVERAGE(), com o custo de percorrer apenas o vetor da solução parcial. No intuito de otimizar as análises, realiza-se uma poda segura, que consiste em parar a análise a partir do momento em que todos os focos forem atingidos, partindo do princípio de que já foi encontrada uma solução candidata. A estratégia proposta foi implementada com custo polinomial, conforme mostra o algoritmo 2.

Algorithm 2 BRUTE-FORCE(D, E)

```
1: bestSolution  $\leftarrow \infty$ 
2: bestVolunteers  $\leftarrow \infty$ 
3: for each  $v_i \in V$  do
4:    $v_{START} \leftarrow v_i$ 
5:    $solutionPartial \leftarrow v_{START}$ 
6:    $volunteers \leftarrow 1$ 
7:   if NOT COVERAGE() then
8:     for each  $v_j \in V$  do
9:        $v_{NEXT} \leftarrow D[v_{START}]$ 
10:       $solutionPartial \leftarrow v_{NEXT}$ 
11:       $volunteers \leftarrow volunteers + 1$ 
12:      if COVERAGE() then
13:         $bestSolution \leftarrow solutionPartial$ 
14:         $bestVolunteers \leftarrow volunteers$ 
15:        BREAK SEARCH
16:      end if
17:    end for
18:  else
19:     $bestSolution \leftarrow solutionPartial$ 
20:     $bestVolunteers \leftarrow volunteers$ 
21:  end if
22: end for
```

1.3 Abordagem de Programação Dinâmica

A estratégia de programação dinâmica é aplicável a problemas nos quais a solução ótima global pode ser computada a partir do cálculo de soluções ótimas locais. Em geral, a grande vantagem dessa abordagem consiste em evitar o recálculo de diversas etapas necessárias na busca pela solução ótima, visto que estes passos foram previamente calculados. Basicamente, é necessário que um problema tenha duas características básicas para aplicar programação dinâmica: subestrutura ótima e superposição de problemas. Neste trabalho, superposição de problemas é evidente na busca pela solução ótima, devido a necessidade de recalcular a abrangência de um voluntário. Por sua vez, nota-se também que encontrar solução ótima global, que minimiza o número de voluntários para acessar todos os focos, está vinculado a encontrar a solução ótima local, de avaliar o vértice que cobre o maior número de focos.

Neste intuito, propomos uma estratégia de programação dinâmica capaz de reaproveitar cálculos anteriores de forma a facilitar a busca pela solução ideal do problema. Especificamente, criamos uma estrutura de dados capaz de armazenar as informações de quantos focos um determinado conjunto de vértices cobre, e avaliamos o processo de busca da solução ótima em etapas de tamanho $|V|$, onde cada uma refere-se ao fato de adicionar um novo vértice aos conjuntos montados. Dessa forma, a estrutura de dados criada é um vetor OPT de tamanho $|1| \times |V^2|$ de forma que cada posição i de OPT refere-se a um conjunto específico de vértices e armazena a quantidade de focos atingidos por este conjunto i . De maneira geral, a estratégia de programação dinâmica segue a recorrência abaixo. Note que, as $|V| + 1$ primeiras posições são referentes aos vértices de D (i.e., vértices resultantes ao caminhar no sentido horário do grafo) e possuem valor 2, uma vez que atingem apenas 2 focos. As demais posições são referentes da união dos conjuntos de vértices das posições $i - |V|$ com $i - (|V| + 1)$, e armazena o número de focos distintos existentes nestes dois conjuntos.

$$OPT_i = \begin{cases} 2 & , i \leq |V| + 1, \\ OPT_{i-|V|} \cup OPT_{i-(|V|+1)} & , i > |V| + 1. \end{cases}$$

O exemplo 2, ilustra a aplicação de nossa estratégia na instância inicial proposta. Note que os $|V| + 1$ elementos de OPT referem-se aos vértices originais no grafo, segundo o vetor D construído pela modelagem proposta. As demais posições de OPT referem-se a união dos conjuntos $i - |V|$ com $i - (|V| + 1)$. Por exemplo, na posição $i = 8$, nota-se a combinação das posições $8 - |V| = 8 - 7 = 1$ com $8 - |V + 1| = 8 - (7 + 1) = 0$, que resulta em 3 focos cobertos devido a interseção dos focos cobertos por V_7 e V_2 . Tal processo se repete até que a solução seja encontrada. Por sua vez, encontramos a solução quando avaliamos cada uma das etapas destacadas a cada $|V|$ posições, conforme o exemplo, e encontramos um valor calculado equivalente ao número de focos da instância, que equivale a 6 no exemplo dado. Tais etapas representam o número de vértices utilizados para as combinações (i.e., etapa 2 possui 2 vértices), e comprovam a premissa de que a primeira solução encontrada sempre será a solução com menor número de vértices. Note que, como uma solução é avaliada sempre em cada etapa do processo, caso sejam encontradas mais de uma solução, tendo assim o mesmo número de vértices, basta realizar o critério dos menores índices para o desempate.

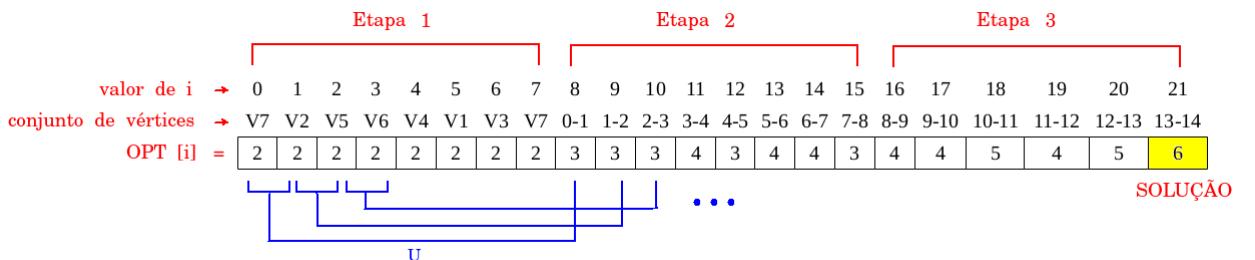


Figura 2: Aplicação da estratégia de programação dinâmica sobre a instância do problema.

O algoritmo 3 demonstra todos os passos realizados para a aplicação dessa estratégia de programação dinâmica. Para representar a estrutura de dados necessária para o processo, utilizamos duas matrizes de dados binárias Z e S , de tamanho $|V^2| \times |F|$ e $|V^2| \times |V|$, respectivamente. Enquanto Z representa os focos de *zika-vírus* cobertos em cada possível solução (linha da matriz), S representa o conjunto de voluntários necessários para a cobertura realizada em cada possível solução (linha da matriz). Conforme ilustrado em 2, a cada iteração o conjunto de vértices e focos unem as posições $i - |V|$ com $i - |V| + 1$, reaproveitando os resultados previamente calculados. O processo se encerra quando todos os focos foram cobertos. A busca pela solução ótima acontece na conclusão de cada etapa, após V iterações, onde verifica-se todos últimos V passos realizados. Em caso de encontrarmos duas ou mais soluções na mesma etapa, o critério de desempate consiste em verificar os índices de cada vértice, optando sempre pela menor soma de índices.

Algorithm 3 DYNAMIC-PROGRAMMING(D, E)

```

1: for  $i = 1$  to  $|V|$  do
2:    $Z_i \leftarrow F1_i \cup F2_i$ 
3:    $S_i \leftarrow D_i$ 
4: end for
5:  $Z_{|V|+1} \leftarrow F1_1 \cup F2_1$ 
6:  $S_{|V|+1} \leftarrow D_1$ 
7:  $v_{ITR} \leftarrow |V| + 1$ 
8: while NOT COVERAGE() do
9:    $v_{START} \leftarrow v_{ITR} - |V|$ 
10:   $v_{NEXT} \leftarrow v_{ITR} - (|V| + 1)$ 
11:   $Z_{v_{ITR}} \leftarrow Z_{v_{START}} \cup Z_{v_{NEXT}}$ 
12:   $S_{v_{ITR}} \leftarrow S_{v_{START}} \cup S_{v_{NEXT}}$ 
13:  if  $v_{ITR} \text{ MOD } |V| == 0$  then
14:    for  $v = (v_{ITR} - |V|) + 1$  to  $v_{ITR}$  do
15:       $\text{Solution} \leftarrow \text{FIND-SOLUTION}(Z_v, S_v)$ 
16:    end for
17:  end if
18: end while

```

1.4 Abordagem Gulosa

A estratégia gulosa é uma técnica utilizada para resolver problemas de otimização. A característica fundamental desta abordagem consiste procura por soluções ótimas locais, na esperança que essas levem a um ótimo global. De maneira geral, a técnica baseia-se em uma função heurística capaz de realizar escolhas locais para a tomada de decisão no momento da procura por soluções. Por basear-se em uma heurística, a técnica nem sempre é capaz de encontrar a solução ótima para um determinado problema. Entretanto, neste trabalho, é proposta uma estratégia gulosa ótima para solucionar o problema ZikaZeroAnelDual devido a uma propriedade relacionada ao formato em anel do grafo.

Para o problema em questão, a solução gulosa proposta consiste em realizar uma série de decisões a respeito da escolha de um determinado vértice para a possível solução. Basicamente, a primeira e principal tarefa a ser realizada consiste em encontrar o vértice inicial da solução. Para tal, pretendemos encontrar o vértice cuja a maior distância para um foco seja a menor em comparação aos demais vértices. Em outras palavras, deve-se calcular as distâncias de todos os vértices para todos os focos por meio de uma busca em largura, e selecionar um conjunto com as maiores distâncias encontradas em cada vértice. A partir desse conjunto, seleciona-se o vértice que possui o menor número de vértices necessários para cobrir todos os focos e também

a menor distância dentre as maiores. Com essa heurística de decisão, o algoritmo está sempre minimizando o número de vértices utilizados na solução, razão pela qual esta é uma estratégia ótima.

Dessa forma, o primeiro passo do algoritmo proposto para essa abordagem consiste em calcular, por meio de uma busca em largura que visa atingir todos os focos de uma dada instância, a distância de todos os vértices para todos os focos e também o número de vértices necessários para tal tarefa. A partir dessa distância calculada e armazenada em uma matriz $DIST$, de tamanho $|V| \times |F|$, seleciona-se os maiores valores de cada vértice (i.e., linha da matriz) e constrói o vetor MAX , de tamanho $|V| \times 1$. Em seguida, seleciona-se o vértice com o menor elemento de MAX e também com o menor número de vértices necessários para cobrir todos os focos, para se tornar o vértice inicial da solução. Note que, o vértice inicial sempre vai pertencer a solução ótima do problema. A figura 3 demonstra um exemplo dessa abordagem.

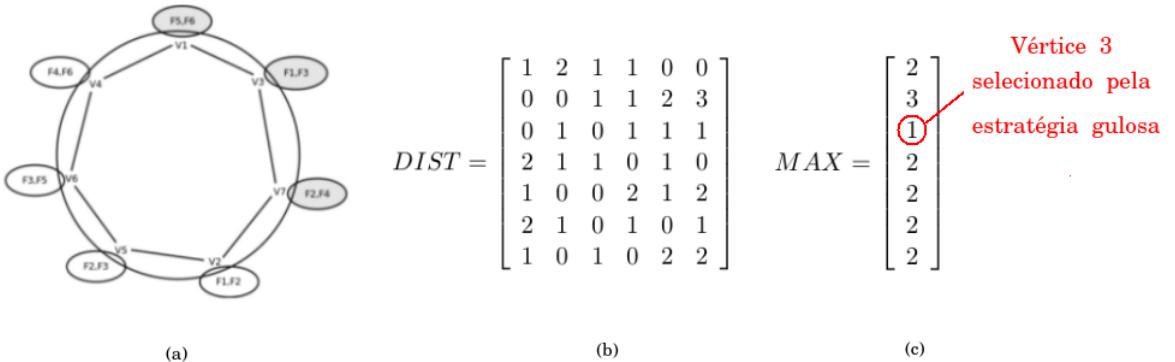


Figura 3: Aplicação da heurística de decisão sobre a instância inicial do trabalho: (a) Grafo anel proposto; (b) distâncias de todos os vértices para todos os focos calculada; (c) seleção do vértice com a menor distância dentre as maiores calculadas.

Em seguida, a partir do vértice inicial selecionado, o algoritmo constrói a solução por meio de escolhas ótimas locais. Basicamente, enquanto a solução parcial não atinge todos os focos do *zika-vírus*, adiciona-se um novo vértice a solução. A escolha desse novo vértice é realizada pela mesma heurística de decisão utilizada na escolha do vértice inicial. Como a cada vértice adicionado na solução as colunas dos focos cobertos são zerados, a heurística sempre vai escolher a menor dentre as maiores distâncias dos vértices da esquerda e da direita da solução parcial. O algoritmo 4 demonstra esse processo.

Com base no algoritmo proposto, podemos notar que a Busca em Largura (linha 3: algoritmo 4) realiza três cálculos importantes para nossas análises. Por meio desta, calculamos: (1) *dist*: a matriz de distâncias de cada vértice para todos focos; (2) *numElements*: um vetor contendo o número de vértices encontrados pela busca em largura até o momento em que todos os focos foram cobertos a partir de um vértice inicial; e (3) *sumIds*: um vetor contendo a soma dos índices dos vértices necessários para cobrir todos os focos. A função SOLUTION-IDEAL() é a responsável por avaliar qual próximo vértice deve ser adicionado a solução. De maneira geral, essa função se preocupa em comparar os valores máximos dos vértices candidatos (vértices a esquerda e a direita da solução parcial). O critério de desempate, é novamente estabelecido pela menor soma dos índices dos vértices candidatos a estarem na solução. Para garantir que a solução sempre encontre o menor número de vértices, também é avaliado os valores contidos no vetor *numElements* obtidos pela busca em largura.

Algorithm 4 GREEDY(D, E)

```
1:  $dist \leftarrow \infty$ 
2: for each  $v \in V$  do
3:    $dist, numElements, sumIds \leftarrow BFS(v)$ 
4: end for
5:  $max \leftarrow MAX(dist)$ 
6:  $v_{START} \leftarrow \{v \in V \mid MIN(max) \cap MIN(numElements)\}$ 
7:  $solution \leftarrow v_{START}$ 
8:  $dist[F1[v_{START}]] \leftarrow 0$ 
9:  $dist[F2[v_{START}]] \leftarrow 0$ 
10:  $v_{ITR} \leftarrow v_{START}$ 
11: while NOT COVERAGE() do
12:    $v_R \leftarrow D[solution]$ 
13:    $v_L \leftarrow E[solution]$ 
14:    $v_{ITR} \leftarrow SOLUTION-IDEAL(v_R, v_L)$ 
15:    $solution \leftarrow v_{ITR}$ 
16:    $dist[F1[v_{START}]] \leftarrow 0$ 
17:    $dist[F2[v_{START}]] \leftarrow 0$ 
18: end while
```

2 Análise da Complexidade Temporal e Espacial

No intuito de analisarmos previamente o desempenho do algoritmo implementado, avaliamos o custo computacional de cada uma das estratégias propostas. De maneira geral, avaliamos o custo de cada procedimento, bem como o custo de armazenamento gerado por cada rotina executada. De maneira geral, como todos os três algoritmos utilizam as funções *COVERAGE()* e *SUMIDS()*, explicitamos que a complexidade temporal delas é $O(|V|)$, por termos que, em ambas, verificar todos os vértices da solução prévia. Por sua vez, a complexidade espacial destas é $O(1)$, visto que não necessitam de nenhuma estrutura de dados para armazenar, mantendo apenas uma variável lógica e um contador.

2.1 Abordagem Força Bruta

A estratégia proposta por essa abordagem consiste em avaliar todas as possíveis soluções existentes a partir de todos os vértices do conjunto V . Tal tarefa, possui custo polinomial de $O(|V|^2)$ dada a necessidade de, no pior caso, percorrer todos os vértices por $|V|$ vezes. Entretanto, como em cada uma dessas $|V|^2$ iterações temos que verificar se a solução cobre todos os vértices, por meio da função *COVERAGE()*, acrescenta-se um custo de $O(|V|)$ comparações. Como a primeira verificação da cobertura ocorre apenas $|V|$ vezes e a segunda verificação ocorre $|V|^2$ vezes, a complexidade temporal dessa abordagem tem custo $O(|V|^3 + |V|^2)$.

Por sua vez a complexidade espacial consiste em um vetor para armazenar os vértices a direita, e outro para os vértices a esquerda, ambos com complexidade $O(|V|)$. De maneira análoga, os vetores utilizados para representar a solução parcial e a solução ótima, tem ambos complexidade $O(|V|)$. Os vetores utilizados para representar os focos tem por sua vez complexidade espacial $O(2|V|) = O(|V|)$. Dessa forma, no limite assintótico, a complexidade espacial dessa abordagem está limitada a $O(|V|)$.

2.2 Abordagem Programação Dinâmica

A estratégia de programação dinâmica, por sua vez, baseia-se na união de conjuntos formados a cada etapa de análise. Primeiramente, preenchemos os valores das $|V| + 1$ posições das nossas estruturas de dados, com custo de $O(|V|)$. Em seguida, procuramos a solução analisando, no

pior dos casos, cada uma das $O(|V|^2)$ iterações por meio da função FIND-SOLUTION que tem custo $O(1)$, por fazer apenas a verificação da solução. A união dos conjuntos é realizada por meio do operador lógico OR , tendo custo $O(1)$. Vale destacar que a estrutura condicional garante que a verificação da solução ocorra apenas nas $|V|$ etapas do processo, como ilustrado em 2, fazendo com que todo o processo iterativo tenha custo apenas de $O(|V|^2)$. Dessa forma, a complexidade temporal tem custo $O(|V|^2 + |V|)$.

A complexidade espacial, está ligada as estruturas de dados utilizadas nessa análise. Como as matrizes Z e S tem tamanho $|V|^2 \times |F|$ e $|V|^2 \times |V|$, a complexidade espacial pode ser expressa por: $O(|V|^2 \times |F| + |V|^2 \times |V|)$.

2.3 Abordagem Gulosa

A estratégia gulosa, conforme mostrado anteriormente, baseia-se na realização de uma busca em largura para calcular as distâncias dos vértices para todos os focos e em seguida, no processo de construção da solução. Apesar das adaptações necessárias para a realização da busca em largura, a complexidade temporal dessa tarefa ainda é $O(|V| + |E|)$, sendo realizada $|V|$ vezes. A complexidade das funções MAX() e MIN() deste processo estão também limitadas a $O(|V| \times |F|)$ e $O(|V|)$, respectivamente, visto que deve-se analisar todo a matriz de distâncias, bem como o vetor resultante da função MAX(). O processo de construção da solução consiste em adicionar vértices à solução até que todos os focos sejam cobertos. No pior dos casos, este processo vai visitar todos os vértices e adicioná-los à solução, com um custo computacional de $O(|V|)$. Neste processo, a escolha dos vértices a direita e a esquerda do vértice inicial tem complexidade temporal de $O(2|V|) = O(|V|)$, visto a necessidade de percorrer todo o vetor D e E . Dessa forma, a complexidade temporal final do algoritmo guloso, no pior dos casos, é:

$$\begin{aligned} f(n) &= O(|V|^2 + |VE|) + O(|V| \times |F|) + O(|V|) + O(2|V| * |V|) \\ &= O(3|V|^2 + V(|E| + |F|) + |V|) \\ &= O(|V|^2 + |V|(|E| + |F|)) \end{aligned} \tag{1}$$

Por outro lado, a complexidade espacial, está novamente sujeita as estruturas de dados utilizadas para armazenar os vértices a direita e a esquerda, bem os vetores relativos aos focos. Dessa forma, no limite assintótico, a complexidade espacial no pior caso é dominada por $O(|V|)$.

3 Implementação do Algoritmo

Os algoritmo propostos para as três abordagens foram implementados 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.

4 Análise de Desempenho

A análise de desempenho realizada consiste em avaliar o tempo de execução das três abordagens, no intuito de comparar a eficiência das distintas estratégias. Primeiramente, calculamos o tempo de execução necessário pelas três abordagens aumentando o número de vértices do grafo de entrada. O gráfico 4 ilustra o comportamento das distintas estratégias. Neste, nota-se a grande vantagem tomada pelo algoritmo guloso com relação aos demais. Por sua vez, nota-se também que o algoritmo de programação dinâmica implementado não é uma abordagem eficiente e portanto, possui o maior tempo de execução necessário.

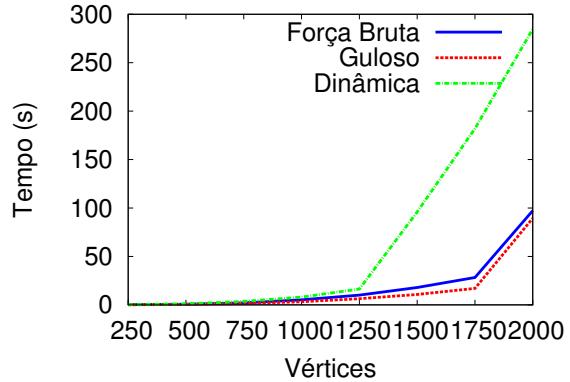


Figura 4: Análise de tempo ao aumentar o número de vértices das instâncias iniciais.

De forma complementar, avaliamos o tempo de execução dos algoritmos ao aumentar o número de focos das instâncias. Na figura 5, que ilustra os resultados obtidos, pode-se notar que o tamanho dos dados iniciais afeta os resultados obtidos. Novamente, o algoritmo de programação dinâmica possui um desempenho mais elevado que os demais. O algoritmo guloso se manteve como o melhor dos resultados.

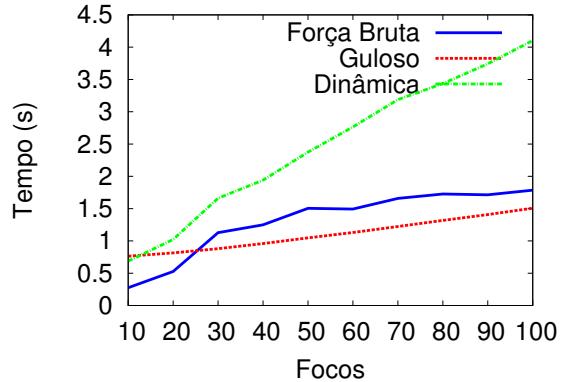


Figura 5: Análise de tempo ao aumentar o número de focos das instâncias iniciais.

5 Comparação dos Resultados

Ao avaliarmos os resultados encontrados, nota-se que o desempenho dos algoritmos estão intimamente relacionados a instância de entrada do problema, bem como as complexidades temporais e espaciais. Com o aumento do número de vértices, arestas ou mesmo focos do problema, o tempo de execução aumenta consideravelmente, refletindo nas complexidades temporais analisadas. Outro fator extremamente relevante para o desempenho dos algoritmos consiste nas complexidades espaciais consideradas. A tabela 1 resume todas as análises feitas na seção 2.

Abordagem	Temporal	Espacial
Força Bruta	$O(V ^3 + V ^2)$	$O(V)$
Programação Dinâmica	$O(V ^2 + V)$	$O(V ^3 + V ^2 \times F)$
Guloso	$O(V ^2 + V (E + F) + V)$	$O(V)$

Tabela 1: Complexidades de cada algoritmo.

O aumento do tempo de execução dos algoritmos propostos, ao aumentarmos o tamanho das instâncias, refletem nas análises de complexidades estudadas. O fato do tempo de execução ter um comportamento muito maior para o algoritmo de programação dinâmica deve-se as estruturas de dados utilizadas. De maneira geral, as estrutura de dados ineficientes aumentam o tempo de execução consideravelmente, devido a alta complexidade espacial do algoritmo de programação dinâmica. Por outro lado, nota-se que o desempenho dos algoritmos de força bruta e guloso refletem apenas suas complexidades temporais, visto que a complexidade espacial destes são equivalentes. Como trabalhos futuros, destaca-se a necessidade de otimizar o algoritmo de programação dinâmica, para que suas vantagens possam ser aproveitadas.

Projeto e Análise de Algoritmos: Trabalho Prático de Paradigmas - ZikaZeroAnelDual

Luiz Henrique Santos
luiz.santos@dcc.ufmg.br

Exercício 1

1. Força-bruta

O primeiro passo foi executar uma busca em profundidade para obter uma ordem dos vértices no anel à partir de um vértice arbitrário. Isso permite buscar a solução apenas em intervalos contíguos $[i,j]$ pois, garantidamente, o grafo nesse intervalo é conexo. Como estamos buscando intervalos distintos num ciclo, concatenamos o vetor de índices com ele mesmo para verificar todos os intervalos $[i,j]$ válidos. Para cada intervalo, verificamos o conjunto de focos que ele cobre, e se a cobertura for total, separamos o intervalo como solução candidata. Mantemos a melhor solução atual a cada iteração, e a cada solução melhor encontrada atualizamos essa solução. Ao verificar todos os intervalos possíveis, sabemos que a solução ao final é ótima.

2. Programação Dinâmica

O primeiro passo é o mesmo descrito na força-bruta: executar uma busca em profundidade para obter uma ordenação dos vértices no anel. Em seguida, seguindo a ideia de concatenação do vetor de índices da força bruta, construímos uma matriz $(2N-1) \times (2N-1)$ e preenchemos sua diagonal principal com as soluções individuais de cada vértice, para em seguida preencher as N diagonais adjacentes. A intuição é que, para o intervalo $[i,j]$, temos duas subsoluções $[i+1,j]$ e $[i,j-1]$. Se ambos os sub-intervalos forem soluções viáveis, a solução de $[i,j]$ é a melhor entre eles. Se apenas uma das soluções é viável, essa é automaticamente a solução para $[i,j]$ pois qualquer outra possível solução com o sub-intervalo que ainda não é solução precisaria de mais vértices, o que configura uma solução pior. Finalmente, se nenhum sub-intervalo é solução, podemos aumentar qualquer um deles pela direita ou pela esquerda em $O(1)$ para construir a solução de $[i,j]$ e verificar se $[i,j]$ é viável.

Como para cada posição na matriz (i.e. intervalo) precisamos da solução à esquerda e da solução abaixo, preenchemos a matriz iterando pelas diagonais (a implementação em C++ modifica a maneira de iterar por questões de eficiência, não utilizando estritamente a notação $[i,j]$ para se referir ao intervalo $[i,j]$) .

Exercício 2

1. Força-bruta

Existem $\frac{n^2+n}{2}$ intervalos distintos, e para cada intervalo iteramos por no máximo n vértices. Como cada vértice tem 2 focos, o número de operações fica limitado a $2n * \frac{n^2+n}{2} = n^3 + n^2$, o que é $O(n^3)$. A complexidade de tempo final ainda é $O(n^3)$, pois a DFS é linear em n . A complexidade assintótica espacial fica limitada ao armazenamento do grafo, $n+m+r = n+n+2n = 4n = O(n)$.

2. Programação Dinâmica

Assim como na solução com força-bruta, verificamos $\frac{n^2+n}{2}$ intervalos distintos. Apesar da matriz ser $(2N-1) \times (2N-1)$, não preenchemos toda a matriz (o que é uma brecha para otimização de espaço, que não foi feita nesta implementação). Para cada intervalo $[i,j]$, ou copiamos a solução de um intervalo menor, ou construímos em $O(1)$ a solução do novo intervalo $[i,j]$. Essa construção em $O(1)$ é possível pois a implementação em C++ utiliza um `unordered_set`, que funciona basicamente como uma tabela hash. O custo temporal de calcular a solução fica então em $O(n^2)$.

O custo espacial já aumenta bastante em relação ao força bruta: cada posição da matriz guarda um `unordered_set`, de tamanho máximo n . A complexidade espacial (custo da matriz mais custo de armazenar o grafo) fica então $n * (2n - 1)(2n - 1) + 4n = O(n^3)$.

Exercício 4

Uma instância com n vértices e $2*n$ focos distintos deveria retornar todos os n :

2 4

4 3

1 3

1 2

8

1 2

3 4

5 6

7 8

Ambos os algoritmos retornam 1 2 3 4 para essa instância. Já uma instância com n vértices e 2 focos, apenas um vértice é necessário:

4 4

2 4

4 3

1 3

1 2

2

1 2

2 1

1 2

2 1

Onde a solução encontrada por ambos é 1.

TRABALHO PRATICO 2

Projeto e Análise de Algoritmos

Paradigmas de Programação

Yuri Pessoa Avelar Macedo
2016662047

Observações

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

G = grafo de entrada
G_o = grafo ótimo
V = conjunto de vértices
E = conjunto de edges (arestas)
R = conjunto de 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.a

Solução por Força Bruta

Dado um grafo anel dual $G(V, E)$ em que cada vértice possui dois de R focos, deseja-se encontrar o menor número de vértices conectados tal que todos os focos são cobertos.

Felizmente, para nossa primeira abordagem, o problema em si é simples o suficiente tal que uma busca por todas as soluções possíveis é de complexidade polinomial, dado a estrutura de anel do grafo. Fixando um vértice v no grafo, podemos obter todos os subgrafos que começam por ele, e se expandem em sentido horário. Dado que os subgrafos que começam por v podem ter tamanho 1 (*apenas v*) até tamanho V (*expande dando uma volta no grafo é cobrindo todos os vértices*) temos que montar todos os subgrafos que se expandem a partir de v tem custo:

$$\sum_{i=1}^V i = V(1 + V) = V^2$$

Cobrindo todos os subgrafos que se iniciam por v , é possível selecionar o de menor tamanho que cobre todos os focos. Entretanto, ainda falta verificar todos os subgrafos que, ou não possuem o vértice v , ou que possuem o vértice v mas que surgem da expansão de outro vértice. Esses grafos podem ser obtidos por realizar o mesmo processo de expansão realizado em v a partir de todos os outros vértices:

$$V * V^2 = V^3$$

Sendo assim, podemos realizar nosso algoritmo de força bruta por este método: para todos os vértices de V , expandimos em sentido horário até que a expansão de cada vértice tenha completado uma volta pelo anel. O menor conjunto encontrado dentre todas as expansões que possui todos os focos é uma solução ótima para o problema.

Tendo em vista que queremos a menor expansão, podemos realizar a seguinte otimização: definimos que todos os conjuntos se expandem ao mesmo tempo, mantendo o mesmo tamanho. Ou seja, primeiro testamos todos os conjuntos com tamanho 1 (*apenas os vértices de inicio*), então todos os conjuntos de tamanho 2 (*o vértice de inicio acrescido do próximo vértice em sentido horário*), etc.

A vantagem é que dessa forma, o primeiro subconjunto que encontrar uma solução válida é garantido de ser ótimo, uma vez que todos os conjuntos de tamanho menor garantidamente não formam uma solução válida para o problema. O que permite que o algoritmo retorne a primeira solução que encontrar e não precise verificar os subconjuntos de maiores tamanhos.

Inicializa $ V $ conjuntos vazios $B_1 \dots V $	Cria um número de conjuntos vazios igual ao número de vértices
Para todo v_i de V :	...
- Adiciona v_i a B_i	Cada conjunto recebe um dos vértices do grafo.
Enquanto $B_1 \dots V \neq V$:	...
- Para todo B_i :	Até que todos os conjuntos tenham todos os vértices...
- Se B_i cobre todos os focos:
- Retorna B_i	Se um dos conjuntos contém todos os focos, então a solução ótima foi encontrada.
- Caso contrário	...
- Adiciona $v_j \in B_i$ a B_i , tal que v_j sucede todos os $v_i \in B_i$ seguindo o grafo em sentido horário.	Caso contrário, adiciona um novo vértice que não está no conjunto. Assim, a cada iteração, todos os conjuntos crescem em 1 vértice (o crescimento de todos os grupos é feito em sentido horário, seguindo o anel). Quando um deles encontrar uma solução que contenha todos os vértices, então essa solução é garantida de ser a solução ótima.
Retorna V	...
	Se nenhum conjunto for encontrado, então a única solução é todos os vértices (dado que o problema é resolvível).

Questão 1.b Solução por Programação Dinâmica

Observando a abordagem bruta, é fácil de perceber o seguinte problema: diversos conjuntos estão sendo calculados mais de uma vez. Suponha um grafo $G(V, E)$, com $|V| = 4$ cujos vértices podem ser renomeados sequencialmente letras (*ou seja, o*

primeiro vértice do grafo é o vértice A, o segundo em sentido horário é o vértice B, seguido de C, e depois D que possui A como vizinho). Este grafo ABCD terá os seguintes subgrafos montados a cada iteração:

	Start = A	Start = B	Start = C	Start = D
I = 1	A	B	C	D
I = 2	AB	BC	CD	DA
I = 3	ABC	BCD	CDA	DAB
I = 4	ABCD	BCDA	CDAB	DABC

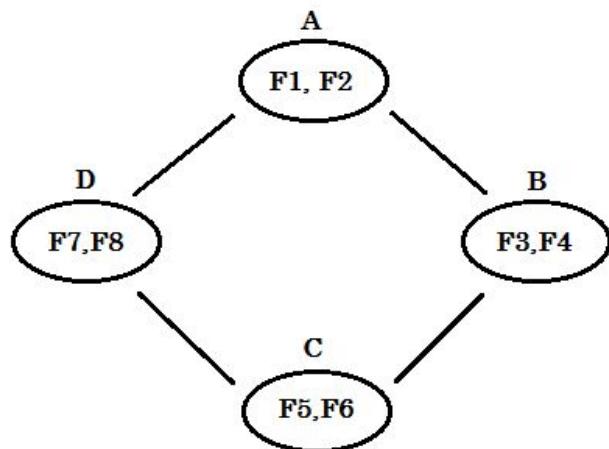
OBS: A montagem de cada subgrafo tem custo igual ao custo de percorrer o grafo até o último vértice do subgrafo. O que gera o custo final $O(V^3)$

A cada subgrafo construído, é necessário percorrer o grafo de forma a obter todos os focos. Na segunda iteração, montar um grafo como AB requer que percorra-se os vértices A e B. Na terceira, montar um grafo como ABC requer que percorra-se os vértices A,B, e C, apesar de que os focos do grafo AB já tenham sido calculados em uma iteração passada. Com isso, define-se o seguinte subproblema:

Subproblema:
Dado um subgrafo G_o de G , gerado a partir de uma expansão de V_1 até V_k , deseja-se determinar quais focos são cobertos pelos vértices V_1 até V_k .

Programação dinâmica nos oferece o recurso de memoização para armazenar o cálculo do subproblema que normalmente é resolvido no tempo de se percorrer os vértices V_1 até V_k , o que é feito em $O(N)$. Observando a tabela, é simples de perceber que determinar os focos de um conjunto de vértices V_1 até V_k pode ser realizado por somar os focos do conjunto com $V_{1 \dots k-1}$ e V_k . O conjunto AB da tabela por exemplo, é determinado pelos focos do conjunto A e do conjunto B. O conjunto ABC é determinado pelos focos do conjunto AB e do conjunto C, e assim por diante.

Utilizando uma tabela, podemos mapear cada conjunto a sua cobertura de focos, e verificar o resultado em $O(1)$, da seguinte forma: estabelece-se uma matriz de variáveis booleanas com dimensões $|V| * (|V| - 1)$ por $|R|$. Cada conjunto de vértices de tamanho I está mapeado para as $I * |V|$ linhas da tabela, exceto o conjunto de vértices de tamanho $I = |V|$, que não existem na matriz (já que corresponde a todos os focos do grafo). Cada foco em R possui uma coluna na matriz, podendo assumir valor verdadeiro ou falso, se o foco existe no conjunto ou não. Segue abaixo um exemplo de problema e sua correspondente tabela:



Index	Group	F1	F2	F3	F4	F5	F6	F7	F8
0	A	1	1	0	0	0	0	0	0
1	B	0	0	1	1	0	0	0	0
2	C	0	0	0	0	1	1	0	0
3	D	0	0	0	0	0	0	1	1
4	AB	1	1	1	1	0	0	0	0
5	BC	0	0	1	1	1	1	0	0
6	CD	0	0	0	0	1	1	1	1
7	DA	1	1	0	0	0	0	1	1
8	ABC	1	1	1	1	1	1	0	0
9	BCD	0	0	1	1	1	1	1	1
10	CDA	1	1	0	0	1	1	1	1
11	DAB	1	1	1	1	0	0	1	1

A partir dessa forma de indexação, verificações são feitas em $O(1)$. assim como a adição de valores na tabela usa dois conjuntos já calculados para fazer a adição em $O(R)$, uma vez que são feitos operadores OR entre cada posição do vetor de focos de cada conjunto. O código em função do algoritmo de força bruta é basicamente o mesmo, fora os cálculos do número de focos de um conjunto de vértices, uma vez que o paradigma de programação dinâmica serve para o problema apenas em função de baratear este custo.

Questão 1.c Solução por Greedy Algorithms

Conforme visto nos outros dois paradigmas, a maior dificuldade do problema encontra-se em determinar se dado vértice v faz parte ou não de um subgrafo ótimo G_o de G . Os dois paradigmas previamente mencionados gastam $O(V)$ tempo para determinar isso, por escolher todos os vértices como pontos de inicio. Se soubéssemos se um vértice pertence ou não a um subgrafo ótimo, então resolver o problema seria resolvível em tempo linear $O(V)$ por expandir o grafo G_o de forma gulosa buscando preencher todos os focos através dos vizinhos de v . Entretanto, não é possível obter essa garantia de forma trivial.

Sendo assim, se quisermos estabelecer uma propriedade de escolha gulosa para o problema, devemos entender que essa propriedade é condicional. Não é possível fixar um vértice como parte da solução sem observar os demais (a não ser no caso trivial em que um dos focos está contido em apenas um dos vértices). Portanto, definimos a seguinte escolha gulosa “condicional” abaixo:

Greedy Choice:

Dado um subgrafo G_G de G , gerado a partir de uma expansão de V_1 até V_k , se V_1 é o vértice de inicio de G_G e G_o , então expandir V_1 até V_k , tal que G_G contenha todos os focos, faz com que $G_G = G_o$.

Nossa escolha diz que queremos expandir V_1 , independente de ele aumentar os focos do conjunto ou não, sobre a premissa de adicionar um vértice ao conjunto faz com que a solução esteja mais próxima de ser valida (*com o pior caso sendo quando adicionar os vértices até completar o conjunto resulta na adição de todo o grafo*). Ou seja, a escolha gulosa deseja maximizar a cada momento o número de focos do conjunto. Estando limitada a expandir apenas no sentido horário, a escolha tem de ser feita adicionando o próximo vértice no sentido horário até que todos os focos sejam cobertos pelo conjunto.

Mesmo assim, esse método encontra o mesmo problema dos outros: não há garantia alguma que o vértice de inicio é de fato o primeiro vértice do conjunto. Para isso, assim como nas outras, o algoritmo escolhe todos os vértices como vértices de inicio. Entretanto, nesse caso a primeira solução encontrada não necessariamente é a ótima, precisando que a solução encontrada pela expansão de cada vértice tenha de ser comparada.

Inicializa $|V|$ conjuntos vazios $B_1 \dots |V|$

Para todo v_i de V :

- Adiciona v_i a B_i

De $i = 1$ até V :

- Enquanto B_i não cobre todos os focos:

- Se B_i cobre todos os focos:

- Armazena B_i

- Caso contrário

- Adiciona $v_j \in B_i$ a B_i , tal que v_j sucede todos os $v \in B_i$ seguindo o grafo em sentido horário.

Retorna o menor B_i

Cria um número de conjuntos vazios igual ao número de vértices

...

Cada conjunto recebe um dos vértices do grafo.

...

Até que o conjunto contenha todos os focos...

....

Se o conjunto contém todos os focos, armazene-o e passe para o próximo conjunto.

Caso contrário, adiciona um novo vértice que não está no conjunto. Assim, a cada iteração, todos os conjuntos crescem em 1 vértice (o crescimento de todos os grupos é feito em sentido horário, seguindo o anel). Quando todos os focos são adicionados, armazena o conjunto (no pior caso, isso ocorre quando o conjunto contém todos os vértices)

...

Compara todos os conjuntos e retorna o menor deles.

O algoritmo segue uma escolha gulosa que é condicional. Uma escolha puramente gulosa que não considera a falta de uma subestrutura puramente ótima do problema não seria capaz de consistentemente encontrar uma solução ótima. Manipular o problema afim de suprir essa condição é a única forma de garantir que o algoritmo guloso não seja uma heurística, visto que o problema em si não possui uma subestrutura ótima (resolver problemas parciais ignorando o resto do grafo não é possível, assim como foi visto nos três paradigmas).

Questão 2.a Complexidade Força Bruta

A complexidade de tempo já foi discutida na **Questão 1.a** e foi demonstrada de ser $O(V^3)$. Um fator a mais a ser considerado, é que a cada solução montada, a verificação dos focos percorre um vetor de focos de tamanho $|R|$, gerando uma complexidade final de $O(V^3R)$. A complexidade de espaço também é bem simples. A função utiliza apenas o grafo original, e um grafo auxiliar no qual os caminhos são construídos. A função também não é recursiva, o que evita de realizar múltiplas cópias redundantes da mesma informação, fazendo com que o custo seja apenas o do grafo original em que cada vértice contém um vetor booleano de tamanho $|R|$ contendo os focos do vértice: $O(VR)$.

Questão 2.b Complexidade Prog. Dinâmica

Por evitar de reconstruir soluções, removemos um custo $O(V)$ da solução por força bruta, o que resulta em uma complexidade de tempo de $O(V^2R)$, o que é consideravelmente melhor para instâncias relativamente grandes. Obviamente, isso nos vem com um custo de espaço com o armazenamento da tabela, que conforme discutido na questão **Questão 1.b**, possui dimensões $|V| * (|V| - 1)$ por $|R|$, consumindo um espaço total de $O(V^2R)$.

Felizmente, a tabela é constituída de variáveis booleanas, utilizando uma estrutura `vector<vector<bool>>` de C++ que possui pouco overhead, o que faz com que o custo de espaço da tabela seja praticamente delimitado por V^2R Bits. No pior caso possível, em que $|R| = 2|V|$ o custo para uma instância de 1000 vértices ainda é 2 Gb para a tabela.

Questão 2.c Complexidade Greedy

O algoritmo guloso nos oferece o melhor dos dois mundos. Ele circula o grafo no máximo uma vez a partir de cada vértice, resultando em tempo $O(V^2R)$, assim como por programação dinâmica. O custo de espaço poderia ser maior do que o de força bruta, caso todas as soluções fossem armazenadas. Entretanto, por armazenar apenas a melhor solução e a solução atual sendo verificada, podemos reduzir o armazenamento de soluções para apenas 2, resultando no mesmo custo assintótico da solução por força bruta $O(VR)$.

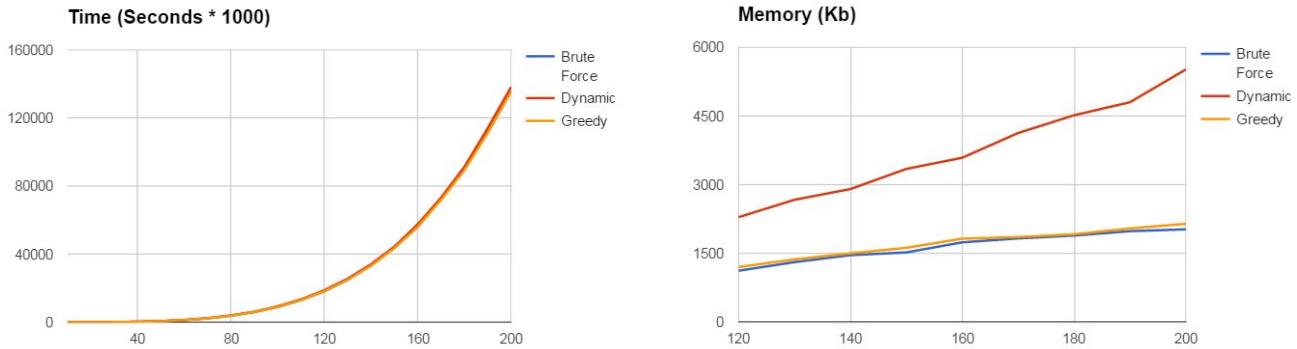
Questão 3 Implementação

O código referente à implementação em C++ do algoritmo para resolver o problema está em anexo a este documento.

Questão 4 Execução

Como os três algoritmos são garantidos de produzirem uma solução ótima, não há sentido em comparar a saída de cada. Para a análise de execução, optou-se por realizar uma análise sobre o tamanho da entrada e o consumo de tempo/espacô de cada solução para instâncias desde o tamanho fornecido pelo enunciado do trabalho (7 vértices), até tamanhos como 200 vértices. Acredita-se que essa bateria de testes nos fornece uma análise bastante precisa dos custos dos algoritmos, nos fornecendo uma boa comparação com as complexidades propostas na **Questão 2**. Para isso, foi criado um gerador de instâncias bastante rudimentar que, dado um número de vértices V , gera uma instância do problema do *ZikaZeroAnelDual* que é o pior caso para os três algoritmos.

Nessas instâncias, o número de focos é $R = 2|V|$ o que faz com que cada vértice possua dois focos únicos que nenhum outro vértice possui. Nesse caso, a única solução para qualquer um dos três métodos é todo o grafo. Para o algoritmo de Programação Dinâmica, vale ressaltar que como a tabela é construída com dimensões sobre variáveis da entrada do problema, o seu tamanho não varia durante a execução. A medição de memória foi feita em Windows com bibliotecas exclusivas desse sistema, e portanto foram tiradas do programa final que deve ser compilado em ambiente Linux.



OBS1: Gráficos de Tempo em função de vértices, e Memória em função de vértices.

OBS2: Para confirmar a óptimalidade dos três métodos, foram feitos testes sobre as instâncias fornecidas no Moodle.

Questão 5 Análise dos Resultados

A função de tempo de complexidade das três são praticamente idênticas e sobrepostas. Com o algoritmo guloso sendo o melhor, seguido do algoritmo de força bruta, e por último o dinâmico, embora essas diferenças sejam praticamente desconsideráveis. A uma primeira impressão poderia indicar problemas na implementação da resolução por programação dinâmica e gulosa.

Feita uma análise mais profunda sobre a complexidade do algoritmo dos algoritmos força bruta e dinâmico, verificou-se todo o código a fim de perceber que elementos consumiam da complexidade do algoritmo. A análise do código está no **Apêndice 1** e **Apêndice 2** deste documento. Esta análise revelou que, devido às operações necessárias para montar a memória do algoritmo, este acaba tendo custo idêntico assintótico, embora as operações de programação dinâmica $O(4V^3)$ ainda sejam um pouco mais custosas que as de força bruta $O(2V^3)$.

A operação de percorrer o grafo, que é a operação barateada por Programação Dinâmica é extremamente simples, dado que o tipo de dado utilizado é um vetor cujos índices são usados para alternar entre elementos. No final a operação é simples suficiente para não valer o investimento, uma vez que outras operações na ordem de $O(V)$ ou $O(R)$ tem de ser feitas para garantir o funcionamento do algoritmo de programação dinâmica. O mesmo vale para as funções do algoritmo guloso: as funções base realizadas em força bruta são leves demais para serem influenciadas pelos benefícios das duas outras metodologias.

Chega-se à conclusão por meio destes testes que o custo de se realizar a verificação e montagem da tabela de programação dinâmica, assim como o armazenamento de outras soluções pelo guloso, não compensam pelo ganho recebido por seu uso durante a implementação.

Apêndice 1 Análise do código Força Bruta

```
string BruteForce(vector<vertex> g, int r){
    ...
    //Order the vector
    do{
        ...
    } while(current != 0); //O(V)
    //Begin
    for (int neighbors_count = 0; neighbors_count < g.size()-1; neighbors_count ++){ //O(V)
        for (int init_count = 0; init_count < g.size(); init_count ++){ //O(V)
            ...
            for(int a = 0; a < g.size(); a ++){solution_chosen[a] = false;} //O(V)
            ...
            for (int i = 1; i <= neighbors_count; i ++){ //O(V)
                ...
            }
            //Check for focuses
            if(Check_Focus(g, r, solution_chosen)){ //O(R)
                ...
                for(int a = 0; a < g.size(); a ++){ //O(V) Acontece apenas 1 vez no algoritmo
                    ...
                }
            }
        }
    }
}
```

```

        }
    }
//Set up the solution
for(int a = 0; a < g.size(); a ++){ //O(V)
    ...
}
return s.str();
}

Complexidade total: O(V) + O(V*V*(V+V+R))+O(V) = O(V^3+R * V^2 )
Pior caso: O(V^3+V^3 ) = O(2V^3)

```

Apêndice 2 Análise do código P.D.

```

string Dynamic(vector<vertex> g, int r){
    ...
    //Order the vector
    do{
        ...
    } while(current != 0); //O(V)
    //Focus Counting
    for (int neighbors_count = 0; neighbors_count < g.size()-1; neighbors_count ++){ //O(V)
        for (int init_count = 0; init_count < g.size(); init_count ++){ //O(V)
            ...
            for(int c = 0; c < v; c ++){solution_chosen[c] = false;} //O(V)
            ...
            for(int i = 0; i < neighbors_count+1; i++){ solution_chosen[order[(init_count+i)%v]] = true;}
        //O(V)
        ...
        int memory_index = (v*neighbors_count)+init_count; //O(V)
        ...
        if(neighbors_count == 0){
            memory_check[memory_index] = true;
            for(int f = 0; f < r; f++){ //O(R)
                ...
            }
        }
        //Check for memory if neighbors > 0
        if(!memory_check[memory_index]){
            ...
            for(int f = 0; f < r; f++){ //O(R)
                ...
            }
        }
        //Read from memory
        for(int f = 0; f < r; f ++){ //O(R)
            ...
        }
        if(Check_Focus(g, r, solution_chosen)){ //O(V) (Chamado apenas uma vez)
            ...
        }
    }
    ...
    return s.str();
}

O(V) + O(V*V*(V+V+R+R) + O(V) = O(2V^3 + 2RV^2)
Pior caso: O(2V^3 + 2RV^2) = O(4V^3)

```

ZicaZeroAnelDual: Implementações Utilizando os Paradigmas de Força Bruta, Programação Dinâmica e Algoritmo Guloso

Átila Martins Silva Júnior¹

¹Departamento de Ciência da Computação - 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 em formato de anel e de forma que todos os focos sejam cobertos pelos voluntários.*

1. Algoritmos

Para resolver o problema do Zica Zero em um grafo anel dual foram propostos três algoritmos, cada um elaborado em um paradigma diferente. Primeiramente, desenvolveu-se um algoritmo no paradigma Força Bruta, em seguida, um algoritmo usando Programação Dinâmica e, por fim, foi elaborado um algoritmo que utiliza estratégia gulosa para selecionar um grupo de voluntários. Todos os algoritmos elaborados possuem Complexidade Assintótica Polinomial, sendo que os que utilizaram Força Bruta e Programação Dinâmica são ótimos, ou seja, sempre encontram a melhor resposta. O algoritmo guloso não encontra a solução ótima sempre, mas alcançou a melhor resposta na maioria dos testes.

1.1. Força Bruta

A ideia do algoritmo por Força Bruta é simples. Começando pelo vértice v_i do anel, percorre os vizinhos de v_i em um único sentido até que todos os focos sejam cobertos. Em seguida, começa pelo vértice v_{i+1} e faz o mesmo processo salvando em P o conjunto de vértices que cobrem todos os focos. Dessa forma, a cada iteração, começa por um vértice diferente e compara o conjunto de vértices encontrado P com a melhor solução até o momento S , se $|P| < |S|$ então o conjunto de vértices mantido em P será transferido para S . Ao final da iteração, quando o algoritmo começou a busca por todos os vértices v_i pertencentes a V , S terá o melhor conjunto de vértices, ou seja, o conjunto do menor número de vértices que cobrem todos os focos.

Abaixo está o pseudocódigo do algoritmo que utiliza Força Bruta para encontrar o número mínimo de voluntários que cobrem todos os focos do mosquito.

Input: A vertex v , a boolean variable $side$ and a ring dual graph G
Output: The set S contain a set starting with the vertex v

```

1  $S \leftarrow S \cup \{v\}$ 
2  $F \leftarrow GET-FOCUS(v)$ 
3 while  $F \neq DISTINCT-FOCUS(G)$  do
4   if  $side = TRUE$  then
5     |  $v \leftarrow v.right$ 
6   else
7     |  $v \leftarrow v.left$ 
8   end
9    $F \leftarrow F \cup GET-FOCUS(v)$ 
10   $S \leftarrow S \cup \{v\}$ 
11 end
12 return  $S$ 
```

Algorithm 1: SET-FROM-DIRECTION

Input: A ring dual graph G
Output: The best set conaining all focuses S

```

1  $v \leftarrow CHOOSE-VERTEX(G)$ 
2  $S \leftarrow SET-FROM-DIRECTION(v, TRUE)$ 
3 for each  $v \in G$  do
4    $P \leftarrow SET-FROM-DIRECTION(v, TRUE)$ 
5   if  $|P| < |S|$  then
6     |  $S \leftarrow P$ 
7   end
8 end
9 return  $S$ 
```

Algorithm 2: BESTSET-BRUTEFORCE

Como se pode observar, algoritmo foi dividido em dois métodos. O primeiro, SET-FROM-DIRECTION, a partir de um vértice v , uma variável booleana $side$ e o grafo dual G encontra um conjunto de voluntários viável, ou seja, que cobre todos os vértices. O segundo método, BESTSET-BRUTEFORCE, varia o valor do parâmetro v para todos os vértices do grafo e, dessa forma, obtém de SET-FROM-DIRECTION os conjuntos de voluntários viáveis e, então, seleciona o conjunto que possui menor quantidade de elementos.

1.2. Programação Dinâmica

O algoritmo do paradigma de Programação Dinâmica, por sua vez, possui um ideia diferente. Ele agrupa os vértices em grupos de tamanho desde um (contendo um vértice cada) e vai aumentando gradativamente o tamanho dos grupos. Desse modo, a cada iteração, é adicionado um vértice vizinho em cada grupo, isso até encontrar um grupo de tenha todos os vértices ou, no pior caso, até formar um grupo com todos os vértices do grafo. Esse algoritmo também garante a solução ótima, uma vez que, quando encontrar o menor conjunto de vértices com todos os focos o algoritmo irá parar e selecionar a melhor solução dentre aquelas que possuem a mesma quantidade de vértices.

Para evitar cálculos repetitivos, esse algoritmo mantém uma matriz com os

cálculos parciais dos focos distintos de cada conjunto de vértices. Dessa forma, ele aproveita os focos distintos de conjuntos de tamanho pequeno para se obter os focos distintos dos conjuntos de tamanho maior. Em seguida, está detalhado o algoritmo descrito em pseudocódigo.

```

Input: A ring dual graph  $G$ 
Output: The best set containing all focuses
1  $v \leftarrow \text{CHOOSE-VERTEX}(G)$ 
2 for  $j \leftarrow 0$  to  $\text{NUM-VERTEX}(G)$  do
3    $covered \leftarrow \text{GET-FOCUS}(v)$ 
4   for  $k \leftarrow 0$  to  $covered.length$  do
5      $M[0][j][covered[k]] \leftarrow \text{TRUE}$ 
6   end
7 end
8  $found \leftarrow \text{FALSE}$ 
9 for  $i \leftarrow 1$  to  $\text{NUM-VERTEX}(G)$  do
10  for  $j \leftarrow 0$  to  $\text{NUM-VERTEX}(G)$  do
11     $count \leftarrow 0$ 
12    for  $k \leftarrow 0$  to  $\text{NUM-DISTINCT-FOCUS}(G)$  do
13      if  $M[0][(j + i) \% \text{NUM-VERTEX}(G)][k] \text{ OR } M[i - 1][j][k]$  then
14         $M[i][j][k] \leftarrow \text{TRUE}$ 
15         $count \leftarrow count + 1$ 
16      end
17      if  $count = \text{NUM-DISTINCT-FOCUS}(G)$  then
18         $found \leftarrow \text{TRUE}$ 
19         $foundPosition \leftarrow i$ 
20         $firstVertex \leftarrow j$ 
21      end
22    end
23  end
24 end
25 for  $i \leftarrow 0$  to  $foundPosition + 1$  do
26    $v = (i + firstVertex) \% \text{NUM-VERTEX}(G)$ 
27    $S \leftarrow S \cup \{v\}$ 
28 end
29 return  $S$ 
```

Algorithm 3: BESTSET-DYNAMIC-PROGRAMMING

O algoritmo BESTSET-DYNAMIC-PROGRAMMING utiliza a matriz M de três dimensões, a primeira dimensão representa os grupos de vértices que pode variar de tamanho um até n , onde n é o número de vértices; a segunda dimensão representa os vértices do grafo que também são exatos n ; e a terceira dimensão representa os focos cobertos por cada grupo, que possui sempre o valor r . Em suma, nesse algoritmo é usada uma matriz $n \times n \times r$.

1.3. Guloso

O algoritmo que usa a estratégia gulosa funciona da seguinte maneira. Inicialmente, é escolhido um vértice qualquer v_i e é selecionado dentre seus vizinhos (v_{i-1} e v_{i+1}) aquele que possui mais focos distintos, esse vizinho então, será adicionado a L . Se ambos os vizinhos de v_i tiverem o mesmo número de focos distintos então será escolhido qualquer um. Nesse caso, L é uma lista, que é mantida de maneira a adicionar sempre na frente os vértices que estão a direita de v_i e atrás, os vértices que estão a esquerda de v_i . Até que se conheça todos os focos, é selecionado dentre os vértices vizinhos, dos vértices que estão na frente e atrás da lista L , aquele que possui mais focos distintos que L . Após encontrar todos os focos, é realizada uma poda em L com o objetivo de remover aqueles vértices de extremidade que possuem focos redundantes. Se acabar a iteração e não se visitou todos os vértices então, começa novamente pelo vértice que ainda não foi visitado salvando a solução em P . Ao final se $|P| < |S|$ então os vértices de S serão substituídos pelos vértices de P . Dessa forma, S terá o conjunto do menor número de vértices dentre os conjuntos analisados.

Diferente dos algoritmos de Força Bruta e Programação Dinâmica, esse algoritmo não garante a solução ótima, uma vez que a escolha gulosa é míope e não garante a melhor escolha global. Por outro lado, a solução será bem próxima da ótima pois o corte realizado após encontrar focos redundantes reduz a quantidade de focos desnecessários. A seguir, está apresentado o pseudocódigo o algoritmo de estratégia gulosa.

Input: A list of vertex L and a set of covered focuses F
Output: The side where is a good neighbor $side$

```

1  $t \leftarrow GET-FRONT(L)$ 
2  $t \leftarrow t.right$ 
3  $FL \leftarrow GET-FOCUS(t)$ 
4  $t \leftarrow GET-BACK(L)$ 
5  $t \leftarrow t.left$ 
6  $FR \leftarrow GET-FOCUS(t)$ 
7  $r \leftarrow GET-DIFF-NEIGHBOR(FR, F)$ 
8  $l \leftarrow GET-DIFF-NEIGHBOR(FL, F)$ 
9 if  $r \leq l$  then
10   |  $side \leftarrow TRUE$ 
11 else
12   |  $side \leftarrow FALSE$ 
13 end
14 return  $side$ 
```

Algorithm 4: GREEDY-CHOICE

Assim como o algoritmo de Força Bruta, para solucionar essa abordagem foram criados dois métodos o GREEDY-CHOICE e o BESTSET-GREEDY. Sendo que o método principal o BEST-SET-GREEDY também utilizou do método SET-FROM-DIRECTION para eliminar os vértices de extremidade que possuem focos redundantes.

O método GREEDY-CHOICE, realiza a escolha gulosa. A partir de uma lista L e o conjunto de focos F cobertos pela lista L , GREEDY-CHOICE escolhe qual vértice vizinho de L possui mais vértices distintos e retorna uma variável booleana indicando de qual lado da lista está esse vértice.

Input: A ring dual graph G
Output: The set of vertex S containing all focuses

```

1  $S \leftarrow GET\text{-}ALL\text{-}VERTEX(G)$ 
2  $v \leftarrow CHOOSE\text{-}VERTEX(v)$ 
3 do
4    $PUSH\text{-}FRONT(L, v)$ 
5    $F \leftarrow F \cup GET\text{-}FOCUS(v)$ 
6    $visited[v] \leftarrow TRUE$ 
7   while  $F \neq DISTINCT\text{-}FOCUS(G)$  do
8     if  $GREEDY\text{-}CHOICE(L, F) = TRUE$  then
9        $t \leftarrow GET\text{-}FRONT(L)$ 
10       $v \leftarrow t.right$ 
11       $PUSH\text{-}FRONT(L, v)$ 
12       $side = TRUE$ 
13    else
14       $t \leftarrow GET\text{-}BACK(L)$ 
15       $v \leftarrow t.left$ 
16       $PUSH\text{-}BACK(L, v)$ 
17       $side = FALSE$ 
18    end
19     $F \leftarrow F \cup GET\text{-}FOCUS(v)$ 
20     $visited[v] \leftarrow TRUE$ 
21  end
22  if  $side = TRUE$  then
23     $P \leftarrow SET\text{-}FROM\text{-}DIRECTION(v, FALSE)$ 
24  else
25     $P \leftarrow SET\text{-}FROM\text{-}DIRECTION(v, TRUE)$ 
26  end
27  if  $|P| < |S|$  then
28     $S \leftarrow P$ 
29  end
30   $v \leftarrow GET\text{-}UNVISITED(visited)$ 
31 while  $ALL\text{-}VISITED(visited) = FALSE$ 
32 return  $S$ 
```

Algorithm 5: BESTSET-GREEDY

O método BESTSET-GREEDY utiliza GREEDY-CHOICE para encontrar, de maneira gulosa, subconjuntos viáveis em G , a partir de então, caso possível, é realizada uma poda nesse subconjunto utilizando SET-FROM-DIRECTION e é escolhido o melhor subconjunto dentre os selecionados. O método BESTSET-GREEDY avalia subconjuntos de G que unidos englobam todos seus vértices, isso não garante que todos os subconjuntos possíveis sejam avaliados. Por isso, embora a solução desse método seja sempre viável, nem sempre será ótima.

2. Análise de Complexidade

Nesta seção será apresentada a complexidade de tempo e espaço dos algoritmos descritos na seção 1.

2.1. Força Bruta

O algoritmo de força bruta, para cada vértice $v \in G$ percorre o grafo anel G em um sentido até encontrar todos os focos. Dessa forma, todos os subconjuntos viáveis são percorridos e, no final, o melhor será escolhido.

2.1.1. Complexidade de Espaço

No grafo anel G cada vértice v possui exatamente duas arestas, uma em cada sentido. Logo, o número de arestas m será duas vezes a quantidade de vértices n . Cada vértice possui também um conjunto de r focos. Logo, como o grafo foi implementado utilizando uma lista duplamente encadeada circular, a função de complexidade de espaço para manter grafo na memória será $S(n, m, r) = n + m + r$. O grafo é utilizado tanto por SET-FROM-DIRECTION quanto por BESTSET-BRUTEFORCE.

Em SET-FROM-DIRECTION, além do grafo, é mantida uma lista S com a solução que pode ter tamanho no máximo n e também uma lista F contendo os focos distintos, com tamanho que varia até r , esse método também utiliza as variáveis v e $side$ que possuem custo um. Logo, o custo espacial de SET-FROM-DIRECTION é dado pela função $S(n, m, r) = 2n + m + 2r + 2$.

Em BESTSET-BRUTEFORCE, além do grafo, é mantida uma lista com a melhor solução até o momento S que tem tamanho no máximo n outra lista P com custo máximo também n e uma variável v que tem custo um. O custo final dessa função será $S(n, m, r) = 3n + m + r + 1$.

A complexidade final de espaço para a estratégia utilizando força bruta é dada pela função $S(n, m, r) = 4n + m + 2r + 3$. Em notação assintótica o custo de espaço será $O(n + m + r)$.

2.1.2. Complexidade de Tempo

A complexidade de tempo foi avaliada levando em consideração o número de comparações entre os elementos. Assim, para obter a complexidade final da abordagem por força bruta é necessário fazer uma avaliação dos dois métodos utilizados.

O método SET-FROM-DIRECTION possui um laço que percorre o grafo até que o conjunto S possua todos os focus. Esse laço executa no máximo n vezes. A condição do *while* possui um custo embutido referente às comparações para verificar se todos os vértices foram cobertos, para cada iteração nessa condição serão realizadas r comparações. Além disso, na interior do laço possui uma comparação para decidir qual lado do grafo seguir. Sendo assim o custo desse método será $T(n, m, r) = n * r$.

Em BESTSET-BRUTEFORCE, primeiramente é escolhido um vértice de G com o método CHOOSE-VERTEX com custo $T(n, m, r) = 1$, em seguida é executada uma vez método SET-FROM-DIRECTION com custo $T(n, m, r) = n * r$. Em seguida, existe um laço que repete n vezes. Para cada repetição é executada uma vez SET-FROM-DIRECTION e é realizada uma comparação para escolher a melhor solução para S . Logo o custo final de BESTSET-BRUTEFORCE é $T(n, m, r) = n^2 * r + n * r + n + 1$. Em

notação assintótica o custo de tempo será $O(n^2 * r)$.

2.2. Programação Dinâmica

O algoritmo que utiliza a estratégia de Programação Dinâmica para encontrar a melhor solução, além do grafo G utiliza uma matriz M para armazenar as soluções parciais de subconjuntos e usá-las posteriormente em conjuntos maiores.

2.2.1. Complexidade de Espaço

Nessa estratégia, além de mantermos o grafo G na memória, é mantida uma matriz M . Como visto na seção 2.1.1, a complexidade de espaço de G é $S(n, m, r) = n + m + r$. A matriz M tem dimensões $n \times n \times r$, então o custo para mante-la na memória será de $S(n, m, r) = n * m * r$. Além disso, são mantidos os vetores S e *covered* ambos com tamanho máximo n e as variáveis *found*, *count*, *foundPosition* e *firstVertex* cada uma com custo $S(n, m, r) = 1$. Logo, a função de complexidade de espaço final é dada por $S(n, m, r) = n * m * r + 3n + m + r + 4$. Em notação assintótica o custo de espaço será $O(n * m * r)$, como se pode observar o custo da matriz M domina sobre o das demais estruturas.

2.2.2. Complexidade de Tempo

Nesse caso, foram consideradas as comparações, exceto as intrínsecas às instruções dos laços de repetição. Como se pode observar no Algoritmo 3, dentro do laço 9 - 24 existem duas comparações na linha 13 que se repetem $(n - 1) * n * r$ vezes e uma fora do laço mais interno que se repete $(n - 1) * n$ vezes, então a função de complexidade de tempo será dada por $T(n, m, r) = 2 * (n - 1) * n * r + n^2$. Assintoticamente o custo é dado por $O(n^2 * r)$.

2.3. Gulosos

O algoritmo que usa a estratégia gulosa para escolher, a cada iteração, o voluntário v que possui maior quantidade de focos distintos de F . Onde F são os focos já cobertos pelo conjunto de amigos L . O voluntário selecionado v deve ser amigo de um dos voluntários presentes em L .

O algoritmo guloso além do método principal BESTSET-GREEDY, utiliza dois métodos auxiliares, o GREEDY-CHOICE que faz a escolha gulosa e o SET-FROM-DIRECTION, mesmo usado no algoritmo de força bruta, nesse caso ele é usado para realizar cortes nos conjuntos que possuem voluntários que cobrem focos que estão cobertos por outros voluntários.

2.3.1. Complexidade de Espaço

Começando pelo método GREEDY-CHOICE, temos que a complexidade de espaço é dada pela lista de vértices L que pode assumir tamanho de no máximo n e a lista de focus F , FL e FR que podem chegar ao tamanho r cada. Ainda existem as variáveis t , r , l e

side que tem tamanho um. O método GET-DIFF-NEIGHBOR conta os vértices diferentes presente no voluntário vizinho, para isso ele utiliza os dois conjuntos de tamanho r e uma variável de retorno. Logo, a função de complexidade de espaço desse métodos é $S(n, m, r) = n + 5r + 5$.

O método BESTSET-GREEDY além de utilizar o grafo G de custo $S(n, m, r) = n + m + r$ utiliza os vetores S , *visited*, L e P de tamanho até n , F que chega a valer r . Possui também as variáveis v , t e *side* cada uma com custo um. O custo de espaço total do algoritmo guloso é $S(n, m, r) = 6n + m + 7r + 8$. Assintoticamente o custo total é da ordem de $O(n + m + r)$.

2.3.2. Complexidade de Tempo

Em GREEDY-CHOICE, as instâncias de GET-DIFF-NEIGHBOR, nas linhas 7 e 9, obtém os focus diferentes nas duas listas passadas por parâmetro com custo custo r cada, em seguida, na linha 9 possui mais uma comparação. Logo, esse método tem custo $T(n, m, r) = 2 * r + 1$.

No método BESTSET-GREEDY temos o laço de repetição das linhas 3 - 31 que só é utilizado quando o laço presente nas linhas 7 - 21 não percorre todos os vértices. Dessa forma, para o cálculo da função de complexidade ire considerar apenas o laço mais interno. Sendo assim, a comparação do laço da linha 7 é realizada n vezes e para cada vez é realizada r comparações adicionais para verificar se os focos foram cobertos, em seguida temos mais uma comparação na linha 8 que é executada a cada iteração do laço. Após o laço, na linha 22 temos mais uma comparação. Na linha 23 temos uma chamada do método SET-FROM-DIRECTION que, como descrito na seção 2.1.2, tem custo $T(n, m, r) = n * r$. Por fim, é realizada uma comparação na lina 27 para escolher a melhor resposta. Em suma o custo de espaço total é dado pela função $T(n, m, r) = 4 * n * r + 2 * n + 2$. Assintoticamente, temos o custo $O(n * r)$.

3. Implementação

A implementação da solução para o problema *ZicaZeroAnelDual* foi realizada na linguagem c++ e segue juntamente com os arquivos deste documento com nome 'ZZAD.cpp'. O programa foi compilado usando o compilador g++ versão 4.8.4 padrão c++11.

4. Testes

Os testes foram realizados com objetivo de comparar o tempo de execução dos algoritmos implementados nos diferentes paradigmas. Sendo assim, cada abordagem foi submetida à mesma entrada e foi avaliado seu tempo de execução. Para que o tempo de execução não seja comprometido pela variação das condições dos ambiente de execução, foram realizadas cinco execuções para cada entrada de um mesmo problema e, em seguida, calculou-se a média dos tempos de execução.

Todos os testes foram realizados em um mesmo ambiente de execução. Processador Intel® Core i3-2310M CPU @ 2.10GHz x 4, 3,8 GiB de memória RAM, rodando o ubuntu 14.04 LTS 64-bit. A execução dos testes foi realizada na mesma seção em um intervalo de tempo contíguo.

Como na definição do problema *ZicaZeroAnelDual* é dado que todo vértice conecta-se a apenas outros dois, então o número de arestas m será sempre proporcional à quantidade de vértices n . Logo, os testes foram realizados considerando apenas o número de vértices n e o número de focos r .

Primeiramente, para avaliar o impacto do crescimento do número de vértices e focos no tempo de execução dos algoritmos, foram criadas instâncias com valores de n e r crescentes em uma mesma razão. Nesse caso, foram realizados testes onde $r = n/2$. Com o resultado da execução dos testes foi gerado o gráfico da Figura 1.

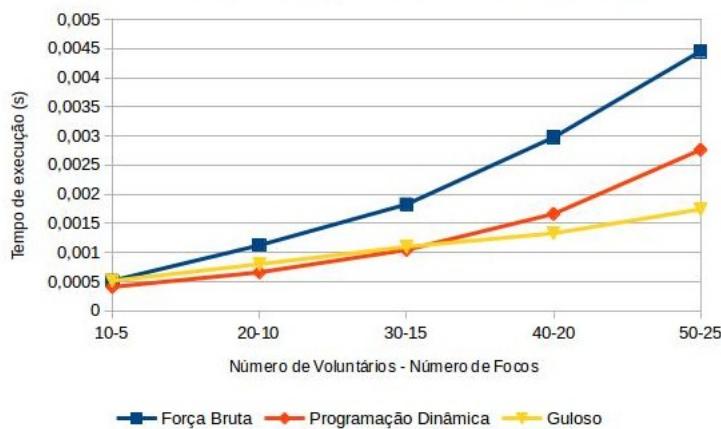


Figure 1. Tempo de execução dos algoritmos com variação no número de vértices e focos.

Como se pode observar na Figura 1, o crescimento da entrada interfere diretamente no tempo de execução de todos os algoritmos. O algoritmo que utiliza força bruta (FB) é o que possui crescimento mais acelerado, isso se deve ao fato de que, nessa abordagem todas as soluções viáveis são testadas para que, então, a melhor solução seja escolhida. O algoritmo que utiliza programação dinâmica (PD) teve desempenho superior ao FB, isso ocorre porque, para chegar a solução ótima, PD não testa todos as soluções viáveis, ele agrupa os vértices até que a solução ótima seja encontrada e, depois, encerra a execução. Por último, o algoritmo guloso (GL) foi o mais rápido para n maior que 30, isso ocorreu pois nessa abordagem abriu-se mão da otimalidade para obter a resposta com mais rapidez.

Nesse caso, quando $r = n/2$, se espaço não for um requisito crítico e não se deseja abrir mão da otimalidade, então o PD será o algoritmo ideal. Por outro lado, quando o espaço é um fator crítico e se deseja uma solução rápida, viável e não ótima, então o algoritmo mais apropriado é o GL. Ainda, se o espaço for um fator limitante e se busca por uma solução ótima, o FB será o mais indicado.

Em seguida, para avaliar o impacto do crescimento do número de focos no tempo de execução dos algoritmos, fixou-se o valor de n e variou-se o valor de r . Com os resultados obtidos pelos testes foi gerado o gráfico da Figura 2.

A Figura 2 mostra que o GL, mesmo com a variação de r ainda apresenta tempos de resposta bem rápidos, se comparado com as outras abordagens. O FB teve um crescimento praticamente linear que é reduzido quando o valor de r se aproxima ao valor de n .

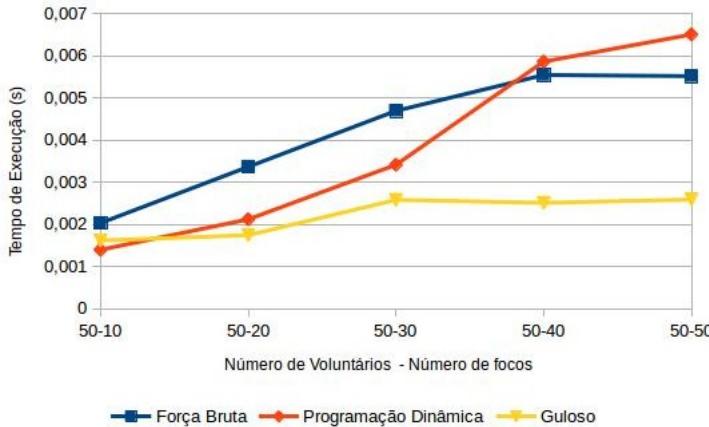


Figure 2. Tempo de execução dos algoritmos com variação no número de focos.

O PD, por sua vez, se mostra bastante lento quando $r \approx n$, demorando mais até que o FB.

Esse resultado nos mostra que quando o número de focos for grande, ou seja, se aproximar do número de voluntários, o algoritmo FB responde mais rápido que o PD. Isso se deve ao fato de que o PD para formar os subconjuntos de vértices, realiza comparações constantes entre os focos de cada vértice em uma matriz de três dimensões. Nesse caso, é preferível aplicar o FB é mais indicado.

5. Análise Vs. Execução

A análise da complexidade de algoritmos tem por objetivo prever o tempo de resposta, avaliar sua eficiência e realizar comparações entre algoritmos. Nesse sentido, para se comparar a análise, teórica, com o tempo de execução prático foram criados gráficos similares aos presentes na Seção 4 mas a partir das funções de complexidade de tempo obtidas nas Seção 2. Na Tabela 1 estão as funções de complexidade das respectivas abordagens seguidas bem como suas análises assintóticas.

Abordagem	Fun. Complexidade	Comp. Assintótica
Força Bruta	$T(n, m, r) = n^2 * r + n * r + n + 1$	$O(n^2 * r)$
Prog. Dinâmica	$T(n, m, r) = 2 * (n - 1) * n * r + n^2$	$O(n^2 * r)$
Guloso	$T(n, m, r) = 4 * n * r + 2 * n + 2$	$O(n * r)$

Table 1. Funções de complexidade dos algoritmos que utilizaram aos paradigmas de Força Bruta, Programação Dinâmica e Guloso.

A partir dessas funções de complexidade foram elaborados os gráficos com os tempos de execução teóricos das Figuras 3 e 4. Nesse caso, o tempo de execução foi obtido substituindo os valores de n e r das funções de complexidade pelo número de vértices e de focos das instâncias de teste.

Quando se compara o gráfico teórico da Figura 3 com o gráfico gerado pelos resultados obtidos pela execução dos algoritmos, Figura 1, pode-se perceber que existe uma ligeira semelhança no comportamento das diferentes abordagens. Pode-se perceber que, assim como na avaliação teórica, o FB tem tempo de execução sempre superior

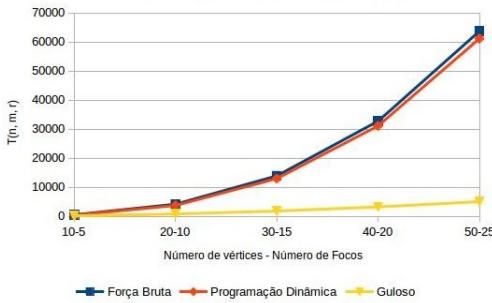


Figure 3. Tempo de execução teórico com variação no número de voluntários e focos.

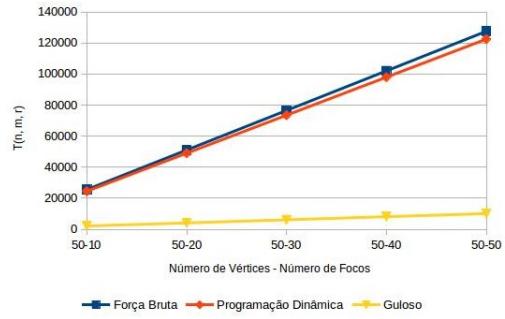


Figure 4. Tempo de execução teórico com variação no número de focos.

aos demais. As curvas do PG e do FB na execução prática seguem um comportamento quadrático presente na curva teórica. Comparando os GL dos dois gráficos pode-se notar que no gráfico do ambiente real o comportamento quase linear presente no gráfico teórico.

Entretanto, a diferença de tempo entre o FB e o PD na Figura 1 se mostra maior que o da Figura 3, isso deve ao fato de que em um ambiente real os processadores realizam otimizações para operações realizadas sobre matrizes e a abordagem PD obtém os resultados a partir de operações realizadas sobre uma matriz.

Com relação ao aumento na quantidade de focus, Figura 4 pode-se notar que a semelhança ocorre até quando $r = 4/5 * n$. Nesse momento, no ambiente real, Figura 2, o custo do PD supera o custo do FB, contrariando a previsão teórica. Por outro lado, nota-se que o comportamento do GL na prática, mais uma vez, se mostra próximo ao teórico.

Por fim, com esses resultados concluiu-se que, uma análise teórica dos tempos de execução dos algoritmos não prevê exatamente o tempo real que esses procedimentos irão executar, mas nos revela a tenéncia do comportamento real desses algoritmos. Nesse trabalho, para calcular a complexidade de tempo considerou-se apenas o número de comparações existentes nos algoritmos, por mais que essa métrica tenha bastante influência no tempo de execução dos algoritmos, para se fazer uma análise ainda mais profunda deve-se considerar outras operações como atribuições, por exemplo. Contudo, os resultados alcançados nesse trabalho foram bastante satisfatórios e possibilitou um amplo entendimento com relação aos paradigmas de programação e suas especificidades.

Trabalho Prático Paradigmas

Bruna V. Frade¹

¹Dept. Ciências da Computação

Universidade Federal de Minas Gerais-(UFMG)

Belo Horizonte– MG – Brasil

brunafrade@c-comp.mest.ufmg.br

1. Exercício 1- Proponha algoritmos de Complexidade Assintótica Polinomial com o Tamanho da entrada (n, m e r) para resolver o Problema ZikaZeroAnelDual

1.1. Força Bruta

Para cada vértice do grafo, verifica todas os sub-grafos não-cíclicos, de forma crescente, que partem deste vértice, ou seja, a partir de um vértice inicial adiciona os focus no conjunto de solução, vai para a adjacência da direita adiciona os focus no conjunto de solução, realiza esse passo até que todos os focus sejam cobertos, fazendo isso para os N vértices.

Quando um sub-grafo atinge todos os focos do problema, ele é salvo como uma solução possível.

Ao realizar isso para cada vértice, teremos N soluções, sendo N o número total de vértices do grafo. Então escolhe-se como solução ótima a solução que contenha a menor quantidade de vértices.

```
1 FUNCAO: forca-bruta()
2     solucoes <- {}
3
4     para cada vertice i do grafo:
5         solucao_corrente <- {}
6         focos_atingidos <- {}
7
8         para cada vertice j do grafo:
9             adiciona o vertice em solucao_corrente
10            adiciona os focos desse vertice em focos_atingidos
11
12            se focos_atingidos possuir todos os focos:
13                adiciona solucao_corrente em solucoes
14
15        solucao <- solucao que possui menor numero de nos
16        retorna solucao
```

1.2. Programação dinâmica

Utiliza-se uma matriz NxN, sendo N o número de vértices do grafo, onde cada coluna representa um vértice e cada linha representa um nível de adjacência. Uma célula[linha][coluna] da nossa matriz possuirá dois campos de valores, sendo o primeiro

uma lista de todos os focos atingidos do vertice 'coluna' até seu nível 'linha' de adjacência, e o segundo o vertice atual daquele nível.

Ex.: Dado o grafo A, U, sendo A o conjunto de arestas e U o conjunto de vertices: A = (1,4),(4,2),(2,3),(3,1) U = 1,2,3,4 com focos: (vertice1= F1,F3),(vertice2= F2,F5),(vertice3= F2,F4),(vertice4= F1,F5) A célula matriz[linha][coluna] com linha = 3 e coluna = 1, teria: O primeiro valor com os focos dos vertices 1 até o nível 3 de adjacência (ou seja, vertices 1, 2, 4) = F1,F2,F3,F5, e o segundo valor seria 2 (é o terceiro vertice a partir do vertice 1) matriz[3][1] = [[1,2,4], 2]

Esta tabela é utilizada de forma que, dado um caminho que comece do vertice i e vá até o vertice j, para descobrir os focos atingidos pelo caminho i até j+1 basta consultar os focos atingidos pelo caminho i até j e somar aos focos do vertice j+1. Dessa forma, evita-se cálculos redundantes, ao custo de uma quantidade maior de informação armazenada. O algoritmo deve então de forma recursiva, iniciando do nível 1 e preencher a matriz até que matriz[linha][coluna] possua um conjunto que contenha todos os focos, sendo a solução consistituida pelo conjunto dos segundos valores de todas as posições de matriz[X,Y] tal que: para X | X = 1,...,linha e Y = coluna.

```

1 FUNCAO: dinamico(n vel)
2   se n vel = 0:
3     para cada vertice N:
4       matriz[n vel][N] <- [focos de N, N]
5       se matriz[n vel][N] possuir todos os focos:
6         retorna [N], N
7   se nao:
8     para cada vertice N:
9       celula <- matriz[n vel -1][N] #conte do da iteracao
10      anterior
11      vertice_atual <- adjacencias[celula[1]]
12      focos_explorados <- celula[0] + focos de N que forem
13        in ditos
14      matriz[n vel][N] <- [focos_explorados, vertice_atual]
15      se matriz[n vel][N] possuir todos os focos:
16        retorna [vertice_atual], N
17
18  lista, ndice = dinamico(n vel)
19  solucao = [lista + matriz[n vel][ ndice ]]
20  retorna solucao, ndice

```

1.3. Algoritmo Guloso

Enquanto não houver uma solução ou que tenha-se percorrido todos os vertices, verifica-se o conjunto de vertices para se explorar, e escolhe aquele que acrescenta uma maior quantidade de focos inéditos. Em caso de mais de um vertice explorar a mesma quantidade de focos inéditos, a escolha entre eles é feita de forma aleatória. Inicialmente a solução está vazia, logo o conjunto de vertices a se explorar é formado por todos os vertices do grafo.

```

1 FUNCAO: guloso()
2   solucao <- { um vertice aleat rio do conjunto de vertices }

```

```

3   focos_atingidos <- { os focos do vertice escolhido }
4   esquerda e direita <- as duas adjacencias do vertice escolhido
5
6   enquanto comprimento(solucao) < total de nos do grafo:
7       ganho_esq <- a quantidade de vertices ineditos que o
8           vertice esquerda possui
9       ganho_dit <- a quantidade de vertices ineditos que o
10          vertice direita possui
11
12      se ganho_dit > ganho_esq:
13          atual <- direita
14          direita <- adjacencia direita de atual
15      se nao:
16          se ganho_esq > ganho_dit:
17              atual <- esquerda
18              esquerda <- adjacencia esquerda de atual
19          se nao:
20              atual <- sorteio entre esquerda e direita
21              atualiza o vertice sorteado com a adjacencia de atual
22
23      adiciona-se o foco atual na solucao
24      adiciona-se os focos ineditos do vertice atual em
25          focos_atingidos
26
27      se focos_atingidos possuir todos os focos:
28          break
29
30  retorna solucao

```

2. Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) dos algoritmos propostos no Exercício 1.

Para Explicar melhor o número de comparações de cada algoritmo segue como exemplo o grafo (figura 1)

Instância ZikaZeroAnelDual – saída.

1 3 7



Figura 1. Exemplo de grafo para explicar os algoritmos

2.1. Força Bruta

Sabendo que nesse algoritmo não existe nenhum método de poda e que sempre testa todas as possibilidades, portanto existe uma iteração n e que para cada vértice dessa n iteração, existe uma sub-iteração que ocorre n vezes, onde n o número total de vértices.

Com base no exemplo da figura 1 temos que o algoritmo realiza todas as comparações sempre, o resultado dessa iteração pode ser melhor entendido com a seguinte tabela (figura 2):

Força Bruta							
Níveis	Nós						
	1	2	3	4	5	6	7
1	(5,6) 1	(1,2) 2	(1,3) 3	(4,6) 4	(2,3) 5	(3,5) 6	(2,4) 7
2	(1,3,5,6) 3	(1,2,3) 5	(1,2,3,4) 7	(4,5,6) 1	(2,3,5) 6	(3,4,5,6) 4	(1,2,4) 2
3	(1,2,3,4,5,6) 7	(1,2,3,5) 6	(1,2,3,4) 2	(1,3,4,5,6) 3	(2,3,4,5,6) 4	(3,4,5,6) 1	(1,2,3,4) 5
4	(1,2,3,4,5,6) 2	(1,2,3,4,5,6) 4	(1,2,3,4) 5	(1,2,3,4,5,6) 7	(2,3,4,5,6) 1	(1,3,4,5,6) 3	(1,2,3,4,5) 6
5	(1,2,3,4,5,6) 5	(1,2,3,4,5,6) 1	(1,2,3,4,5,6) 6	(1,2,3,4,5,6) 2	(1,2,3,4,5,6) 3	(1,2,3,4,5,6) 7	(1,2,3,4,5,6) 4
6	(1,2,3,4,5,6) 6	(1,2,3,4,5,6) 3	(1,2,3,4,5,6) 4	(1,2,3,4,5,6) 5	(1,2,3,4,5,6) 7	(1,2,3,4,5,6) 2	(1,2,3,4,5,6) 1
7	(1,2,3,4,5,6) 4	(1,2,3,4,5,6) 7	(1,2,3,4,5,6) 1	(1,2,3,4,5,6) 6	(1,2,3,4,5,6) 2	(1,2,3,4,5,6) 5	(1,2,3,4,5,6) 3

cada célula = (focos), nó atual Vermelho = solução ótima cálculos do força bruta Negrito = ótimo partindo daquele nó

Figura 2. Tabela de comparações do Algoritmo Força Bruta

Dessa forma, analisando a complexidade de tempo no melhor caso e no pior caso

$$O(n^2)$$

Sabendo que nesse algoritmo teremos sete conjunto de soluções diferentes sendo armazenadas e que para cada um desses conjuntos de soluções, pode ser armazenado até

n vértices, além do custo de armazenamento dos focos descobertos que seriam sempre f . Dessa forma a complexidade de espaço no melhor caso e no pior caso ($n^2 + f$), porém sabendo que $f < n$ podemos expressar a seguinte notação assintótica.

No melhor caso e no pior caso

$$O(n^2)$$

2.2. Programação Dinâmica

Para auxiliar na explicação do número de comparações realizada por esse algoritmo levaremos em consideração o exemplo do grafo da figura 1 e a tabela resultante da figura 3. Nesse algoritmo temos uma matriz de armazenamento estruturada exatamente como a figura 3, dessa forma o algoritmo realiza sempre n comparações para calcular o valor de cada linha, sendo n o número de vértices. Essas n comparações podem ser realizadas $f/2 \dots n$, onde f é o número total de focos e $f/2$ é o número mínimo de vértices para cobrir todos os focos.

		Dinâmico						
		Nós						
		1	2	3	4	5	6	7
Níveis	1	(5,6) 1	(1,2) 2	(1,3) 3	(4,6) 4	(2,3) 5	(3,5) 6	(2,4) 7
	2	(1,3,5,6) 3	(1,2,3) 5	(1,2,3,4) 7	(4,5,6) 1	(2,3,5) 6	(3,4,5,6) 4	(1,2,4) 2
	3	(1,2,3,4,5,6) 7	(1,2,3,5) 6	(1,2,3,4) 2	(1,3,4,5,6) 3	(2,3,4,5,6) 4	(3,4,5,6) 1	(1,2,3,4,5) 5
	4	(1,2,3,4,5,6) 2	(1,2,3,4,5,6) 4	(1,2,3,4) 5	(1,2,3,4,5,6) 7	(2,3,4,5,6) 1	(1,3,4,5,6) 3	(1,2,3,4,5,6) 6
	5	(1,2,3,4,5,6) 5	(1,2,3,4,5,6) 1	(1,2,3,4,5,6) 6	(1,2,3,4,5,6) 2	(1,2,3,4,5,6) 3	(1,2,3,4,5,6) 7	(1,2,3,4,5,6) 4
	6	(1,2,3,4,5,6) 6	(1,2,3,4,5,6) 3	(1,2,3,4,5,6) 4	(1,2,3,4,5,6) 5	(1,2,3,4,5,6) 7	(1,2,3,4,5,6) 2	(1,2,3,4,5,6) 1
	7	(1,2,3,4,5,6) 4	(1,2,3,4,5,6) 7	(1,2,3,4,5,6) 1	(1,2,3,4,5,6) 6	(1,2,3,4,5,6) 2	(1,2,3,4,5,6) 5	(1,2,3,4,5,6) 3

cada célula = (focos), nó atual
Vermelho = solução ótima
cálculos do dinâmico
Negrito = ótimo partindo daquele nó

Figura 3. Tabela de comparações do Algoritmo de Programação Dinâmica

Dessa forma, temos que a complexidade de tempo no melhor caso é

$$\Omega(n(f/2))$$

e no pior caso

$$O(n^2)$$

O pior caso é uma raridade acontecer, pois é necessário que cada vértice consiga cobrir apenas um novo foco, isso começando por qualquer vértice, ou seja, para qualquer conjunto de solução s , temos que $s = n$. Então temos que a complexidade é quase sempre melhor que o pior caso.

Temos que a complexidade de espaço no melhor caso é

$$\Omega(s * n^2)$$

, onde s é o número de vértices que pertence a solução e no pior caso

$$O(n^3)$$

, onde teríamos uma matriz $n \times n$ onde cada célula armazena uma lista de n elementos.

2.3. Algoritmo Guloso

A partir do vértice escolhido para começar, esse algoritmo realiza n iterações, onde cada iteração tenho uma sub-iteração analisando as adjacências desse determinado i , sabendo que o grafo é cíclico então cada vértice tem sempre duas adjacências, portanto o número total de iterações é no máximo n , e no mínimo $(f/2)$ onde f é o número de focos.

Para entender melhor o número de comparações do algoritmo analisemos a tabela da figura 4 que representa uma ilustração das comparações do algoritmo a partir do grafo da figura 1. Para esse exemplo o algoritmo se iniciou do vértice 5 e conseguiu alcançar todos os focos com apenas 5 vértices, realizando dessa forma apenas 5 iterações que é menor que o número total de vértices n que nesse caso $n = 7$.

Guloso								
Iteração	Nós e { focos }							
	0	1 {5,6}	3 {1,3}	7 {2,4}	2 {1,2}	5 {2,3}	6 {3,5}	4 {4,6}
	1	1 {5,6}	3 {1,3}	7 {2,4}	2 {1,2}	5 {2,3}	6 {3,5}	4 {4,6}
	2	1 {5,6}	3 {1,3}	7 {2,4}	2 {1,2}	5 {2,3}	6 {3,5}	4 {4,6}
	3	1 {5,6}	3 {1,3}	7 {2,4}	2 {1,2}	5 {2,3}	6 {3,5}	4 {4,6}
	4	1 {5,6}	3 {1,3}	7 {2,4}	2 {1,2}	5 {2,3}	6 {3,5}	4 {4,6}
	5	1 {5,6}	3 {1,3}	7 {2,4}	2 {1,2}	5 {2,3}	6 {3,5}	4 {4,6}
	6	1 {5,6}	3 {1,3}	7 {2,4}	2 {1,2}	5 {2,3}	6 {3,5}	4 {4,6}
	7	1 {5,6}	3 {1,3}	7 {2,4}	2 {1,2}	5 {2,3}	6 {3,5}	4 {4,6}

Nó explorado na iteração
Nós já explorados

Figura 4. Tabela de comparações do Algoritmo Guloso

Dessa forma, temos que a complexidade de tempo no melhor caso é

$$\Omega((f/2))$$

e no pior caso

$$O(n)$$

Na complexidade de espaço, temos um custo de armazenamento de apenas um conjunto de solução podendo conter $(f/2)$ vértices ou n vértices, além de armazenar o número de focos descobertos a cada iteração, dessa forma, no melhor caso temos $(f/2)+f$ e no pior caso $n+f$. Sabendo que $f < n$ podemos expressar a complexidade da seguinte forma em notação assintótica.

Temos que a complexidade de espaço no melhor caso é

$$\Omega(f)$$

e no pior caso

$$O(n)$$

3. Exercício 3- Implementação

Os algoritmos implementados estão em anexo juntamente com esse documento.

4. Exercício 4- Execute testes da implementação do Exercício 3, propondo instâncias interessantes para o Problema ZikaZeroAnelDual. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n, m e r) comparando os três algoritmos implementados

Ao executar os três algoritmos para diferentes tamanhos de instâncias obtemos o gráfico da figura 5. Para pequenas instâncias já é possível notar a diferença do tempo de execução entre o força bruta e os demais algoritmos.

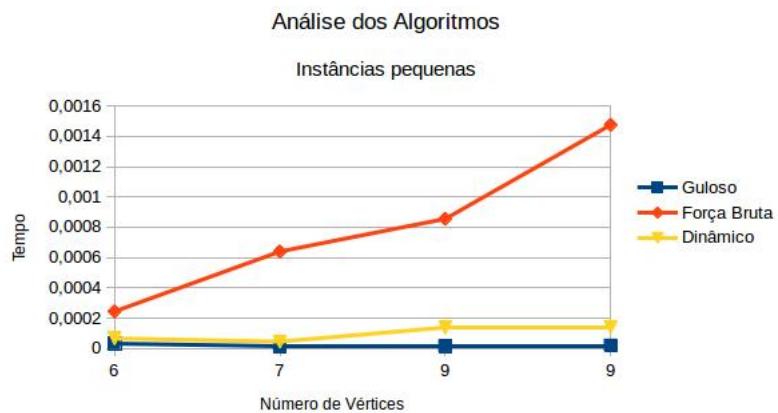


Figura 5. Gráfico com instâncias pequenas

No gráfico da figura 6 podemos notar o quanto o algoritmo de força bruta cresce em tempo com o tamanho da instância de 40 vértices enquanto os outros dois algoritmos estão muito próximos ao tempo 0.

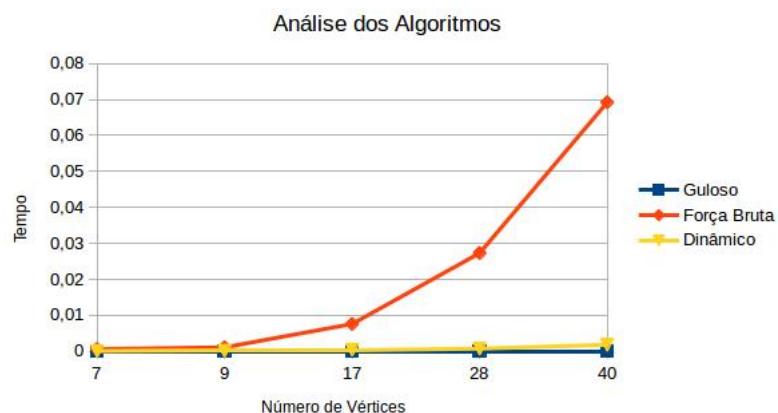


Figura 6. Gráfico medindo desempenho dos três algoritmos com até 40 vértices

No Gráfico da figura 7 temos uma análise apenas dos algoritmos guloso e dinâmico variando o tamanho de instâncias de 6 a 40. Pode ser notado que o dinâmico tem um crescimento muito maior que o algoritmo guloso.



Figura 7. Gráfico de comparação entre Guloso e Dinâmico com até 40 vértices

5. Exercício 5- Compare a análise e a execução, respectivamente, Exercícios 2 e 4. Principal- mente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

De acordo com a análise de complexidade definida no exercício 2, temos que $\text{tempo}(\text{Guloso}) < \text{tempo}(\text{Dinamico}) < \text{tempo}(\text{ForcaBruta})$, pois sabendo que o tempo do guloso no pior caso é n e no dinâmico é $s*n$ sendo s a quantidade de vértices necessários para a solução sendo $s < n$, e temos no força bruta n^2 , portanto $n < (s*n) < n^2$.

De acordo com os gráficos das figuras 5, 6 e 7, temos o tempo de execução na mesma ordem do que foi definido no exercício 2, ou seja $\text{tempo}(\text{Guloso}) < \text{tempo}(\text{Dinamico}) < \text{tempo}(\text{ForcaBruta})$, portanto podemos concluir que a previsão realizada na análise tem resultado equivalente com os tempos obtidos na execução do algoritmo no exercício 4.

Relatório do trabalho prático 2 - Paradigmas

Rodrigo Lemos Cardoso

¹Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

rodrigolc@dcc.ufmg.br

1. Algoritmos

1.1. Força Bruta

Usando a modelagem do trabalho prático 1 para construir o grafo G com voluntários V e focos F , o algoritmo força bruta funciona basicamente da seguinte maneira:

Algorithm 1 ZikaZeroAnelDual Força Bruta

```
1: procedure BRUTE FORCE( $G, V, F$ )  $\triangleright$  Retorna o menor subconjunto de voluntários
2:    $best \leftarrow \{1..|V|\}$   $\triangleright$  A melhor solução até agora e usar todos os voluntários
3:   for  $i \in \{0, \dots, |V| - 1\}$  do  $\triangleright$  Testaremos soluções de tamanho  $i+1$ 
4:     for  $j \in \{1, \dots, |V|\}$  do  $\triangleright$  Testaremos soluções começando em  $j$ 
5:        $candidate \leftarrow try\_solution(j, i)$   $\triangleright$  Chama  $j$  mais os próximos  $i$  vizinhos
6:       if  $best > candidate$  then
7:          $best \leftarrow candidate$ 
8:       end if
9:     end for
10:   end for
11:   return  $best$ 
12: end procedure
```

O algoritmo basicamente testa todas soluções de tamanho i de 1 até $|V|$. Para isso ele testa para cada voluntário j mais os próximos $i - 1$ vizinhos.

A função `try_solution` verifica se o candidato cobre todos os focos F e o operador `<` compara o tamanho do conjunto criado e a ordem lexicográfica do mesmo.

Observação: Foi omitido aqui, mas quando uma solução de tamanho i é encontrada não é necessário testar soluções de tamanho $i + 1$.

1.2. Dinâmico

O algoritmo dinâmico tem a configuração parecida com o algoritmo força bruta, com a diferença de que usa-se memoização. Esta memoização se dá da seguinte maneira.

Algorithm 2 ZikaZeroAnelDual Dinâmico

```
1: procedure DYNAMIC( $G, V, F$ )       $\triangleright$  Retorna o menor subconjunto de voluntários
2:    $best \leftarrow \{1..|V|\}$            $\triangleright$  A melhor solução até agora é usar todos os voluntários
3:    $M_1 \leftarrow best\_solutions(1)\}$      $\triangleright |V|$  melhores soluções de tamanho 1 para cada  $v$ 
4:   for  $i \in \{2, \dots, |V|\}$  do            $\triangleright$  Testaremos soluções de tamanho i
5:     for  $j \in \{1, \dots, |V|\}$  do        $\triangleright$  Testaremos soluções começando em j
6:        $candidate \leftarrow M_{i-1,j} \cup M_{1,(j+i-1)mod|V|}$ 
7:       if  $best < candidate$  then
8:          $best \leftarrow candidate$ 
9:       end if
10:      end for
11:    end for
12:    return  $best$ 
13: end procedure
```

O passo mais capcioso está na linha 6, onde para avaliar uma solução de tamanho i que começa do voluntário j , unimos a solução de tamanho $i - 1$ de j com a solução de tamanho 1 para $j + i - 1$ módulo $|V|$.

Por exemplo, suponha que estejamos avaliando a solução de tamanho 4 do voluntário 3 em um conjunto de voluntários 1,2,3,4,5.

A solução de tamanho 3 engloba os voluntários 3,4,5 pois são $i - 1$ voluntários a frente de 3. Já a solução de tamanho 4 englobará 3,4,5,1, que é uma volta no anel contemplando o voluntário 1 e sua respectiva solução de tamanho 1.

1.3. Gulosos

O algoritmo gulosos é o mais simples. Ele funciona da seguinte maneira:

Algorithm 3 ZikaZeroAnelDual Gulosos

```
1: procedure GREEDY( $G, V, F$ )  $\triangleright$  Tenta retornar o menor subconjunto de voluntários
2:    $best \leftarrow \{1..|V|\}$            $\triangleright$  A melhor solução até agora é usar todos os voluntários
3:   for  $v \in V$  do            $\triangleright$  Testaremos para cada voluntário
4:      $candidate \leftarrow v$ 
5:     for  $w \in candidate.neighbors$  do     $\triangleright$  Entre os vizinhos da solução candidata
6:        $candidate \cup best\_neighbor$          $\triangleright$  Melhor vizinho  $\cup$  solução candidata
7:       if  $best > candidate$  then
8:          $best \leftarrow candidate$ 
9:       end if
10:      end for
11:    end for
12:    return  $best$ 
13: end procedure
```

Este algoritmo gulosos funciona iterativamente. Para cada voluntário o algoritmo

compara sua vizinhança e adiciona o vizinho que cobrir mais focos descobertos e assim sucessivamente gerando um conjunto de voluntários cujos vizinhos em cada ponta do anel são os melhores localmente. No final retorna a solução que obteve o melhor resultado.

2. Complexidade

2.1. Força Bruta

O algoritmo força bruta tem a complexidade assintótica temporal dada pelas 3 iterações em cadeia que ele chama: Uma para o tamanho variável da solução, outra para cada voluntário e outra para avaliar os vizinhos desse voluntário, resultando em uma ordem de complexidade $O(|V|^3)$. A complexidade espacial é $O(|V|)$ pois armazena-se apenas dois vetores de tamanho $|V|$.

2.2. Dinâmico

O algoritmo dinâmico tem complexidade assintótica dada pelas 2 iterações mais uma função logarítmica de inserção em um *set*. Isto pode ser constatado assim: Uma iteração de tamanho de solução cascataada com uma iteração para cada voluntário. Dentro da iteração de avaliar o voluntário j , une-se o conjunto $M_{i,j}$ ao conjunto de dois elementos $M_{0,j+i-1}$. É um número constante de operações logarítmicas, portanto: $O(|V|^2 \log(|V|))$

A complexidade espacial é da ordem de $O(|V|^2)$ pois no pior caso guardaremos cada subconjunto.

2.3. Guloso

O algoritmo guloso itera para cada voluntário e busca localmente bons voluntários próximos a ele até gerar uma solução viável. A união com o melhor vizinho pode ser feita em tempo logarítmico, resultando em $O(|V|^2 \log(|V|))$ com complexidade assintótica espacial de $O(|V|)$ para guardar a melhor solução.

3. Implementação

Os algoritmos foram implementados na linguagem de programação C++ (11) e estão anexados juntos a este documento.

TRABALHO PRÁTICO 2: ZIKAZEROANELDUAL

LUCAS J. C. MACIEL
MATRÍCULA: 2016661733

RESUMO. Este trabalho relata a solução do problema proposto ZikaZeroAnel-Dual, com abordagem aos paradigmas de força-bruta, programação dinâmica e algoritmos gulosos. Foram feito análises assintótica de tempo e espaço de cada solução proposta. Experimentos computacionais foram realizados verificando a corretude e o comportamento dos algoritmos propostos para diversas instâncias automaticamente geradas do problema.

1. MODELAGEM

Podemos descrever o problema proposto com uma modelagem em Grafos, exatamente como no trabalho prático anterior. Seja $G = (V, E)$ um grafo, seja R um conjunto de chaves (no problema, focos), definimos como $\mathcal{R}(v_i) \subseteq R$ uma atribuição de chaves para cada vértice $v_i \in V$. O problema resume em encontrar um subgrafo **conexo** $G' = (V', E')$ de G com $V' \subseteq V$, $E' \subseteq E$ tal que $\bigcup_i^{|V'|} \mathcal{R}(v'_i) = R$ (restrição de cobertura), e que $|V'|$ seja mínimo. Neste trabalho porém, é garantido que o grafo $G = (V, E)$ é sempre um grafo anel, ou seja, um grafo conexo, com $|V| = |E|$ e cada vértice de V possui grau igual a 2.

A adição da restrição de que o grafo G é sempre um grafo anel, transforma o problema original NP-difícil em um problema P, com solução polinomial, a qual veremos na próxima seção.

2. ABORDAGEM POR PARADIGMAS

Nesta seção iremos propor soluções baseadas nos três paradigmas da computação: força-bruta, programação dinâmica e algoritmos gulosos. Iremos identificar as características do problema, que levam as respectivas soluções, como *estrutura ótima, escolha gulosa e sobreposição de subproblemas*.

2.1. Força-Bruta. A solução por força-bruta foi obtida enumerando todo conjunto de soluções, isto é, para cada grafo solução $G' = (V', E')$ possível, verificamos se G' é conexo e se $\bigcup_i^{|V'|} \mathcal{R}(v'_i) = R$ satisfaz a restrição de cobertura. A solução ótima é aquela onde $|V'|$ é mínimo.

Enumerar todos os grafos possíveis pode ser realizado em $O(2^{|V|})$, e verificar se as restrições são atendidas em $O(|V| + |R|)$, obtendo um custo assintótico de execução de $O(2^{|V|}(|V| + |R|))$. O custo em memória é apenas para manter o grafo, que é $O(|V| + |E|)$.

Date: 18 de junho de 2016.

2.2. Programação Dinâmica. Inicialmente, ordenamos os vértices de V em sentido horário (ou anti-horário, como preferir), em uma lista infinita P , tal que cada posição $p_j \in P$ representa um vértice v_i de V , sendo que os vértices com posições p_{j-1} e p_j são adjacentes em G tal como p_j e p_{j+1} . Assim, podemos garantir que um subgrafo $G' = (V', E')$ de G é conexo se e somente se existe uma subsequência $P' = (p_i, p_{i+1}, \dots, p_{j-1}, p_j)$ de P tal que todos os vértices de V possuem posições em P' . É fácil ver que o conjunto solução G' forma um grafo linha conexo, descrito pela lista P .

O nosso objetivo é encontrar posições p_i e p_j com $i \leq j$ e $p_i, p_j \in P$, tal que a restrição de cobertura $\bigcup_{k=i}^j \mathcal{R}(p_k) = R$ seja satisfeita e que o número de vértices $j - i + 1$ seja o menor possível. Podemos a partir daí, encontrar uma subestrutura ótima do problema. Seja $R' \subset R$, dizemos que $dp(i, j, R)$ é 1 se o grafo linha induzido pela subsequência de P entre p_i e p_j satisfaz a restrição de cobertura. Podemos obter a solução para problemas menores, onde a cobertura é um conjunto $R' \subset R$ para subsequências variadas de P . Portanto, dp é a subestrutura ótima onde existe sobreposição de subproblemas tanto em R como em P . Podemos obter dp usando a recorrência:

$$(1) \quad dp(i, i, R') = \begin{cases} 1 & \text{se } \mathcal{R}(p_i) = R' \\ 0 & \text{caso contrário} \end{cases}$$

$$(2) \quad dp(i, j, R') = \max(dp(i+1, j, R' - \mathcal{R}(p_i)), dp(i, j-1, R' - \mathcal{R}(p_j)))$$

O caso base é obtido aplicando a definição de dp para segmentos de tamanho 1. Já a recorrência podemos obter expandindo:

$$\begin{aligned} dp(i, j, R') = 1 &\text{ implica que} \\ \bigcup_{k=i}^j \mathcal{R}(p_k) &= R' \\ \mathcal{R}(p_i) \cup \bigcup_{k=i+1}^j \mathcal{R}(p_k) &= R' \\ \bigcup_{k=i+1}^j \mathcal{R}(p_k) &= R' - \mathcal{R}(p_i) \text{ implicando} \\ dp(i+1, j, R' - \mathcal{R}(p_i)) &= 1 \end{aligned}$$

Fazendo o mesmo para o segundo caso. Assim, podemos encontrar a solução ótima $dp(i, j, R)$ para todo $p_i, p_j \in P$ resolvendo seus subproblemas. Note que não é necessário resolver para todo p_i, p_j , pois a distância máxima entre i e j é $|V|$, e não é necessário calcular posições p_i de um mesmo vértice.

Assim, como é $O(1)$ para calcular um estado, o custo assintótico de execução se dá ao calcular todos os estados para encontrar aquele com $j - i + 1$ mínimo. Dessa forma, como podemos considerar $|P| = 2|V|$, temos $O(|V|) \cdot O(|V|) \cdot O(2^{|R|}) = O(|V|^2 \cdot 2^{|R|})$. O custo de memória é o mesmo pois é necessário armazenar todos os estados.

2.3. Algoritmo Guloso. Algoritmos gulosos são empregados em problemas, que além de possuir uma subestrutura ótima, precisam de possuir uma escolha gulosa, ou seja, não é necessário resolver todos os subproblemas para obter o ótimo global, o ótimo local é também um ótimo global. Verificamos esta propriedade no nosso problema, generalizando a expansão feita na seção anterior:

$$\begin{aligned} dp(i, j, R') = 1 &= dp(i, j, R') \text{ implica que} \\ \bigcup_{k=i}^j \mathcal{R}(p_k) &= R' = \bigcup_{k=i}^j \mathcal{R}(p_k) \\ \mathcal{R}(p_i) \cup \bigcup_{k=i+1}^j \mathcal{R}(p_k) &= R' = \bigcup_{k=i}^{j-1} \mathcal{R}(p_k) \cup \mathcal{R}(p_j) \text{ implicando a relação} \\ dp(i+1, j, R' - \mathcal{R}(p_i)) &= dp(i, j-1, R' - \mathcal{R}(p_j)) \end{aligned}$$

Assim, a equação de recorrência pode ser reescrita como:

$$(3) \quad dp(i, j, R') = dp(i+1, j, R' - \mathcal{R}(p_i)) = \begin{cases} 1 & \text{se } \bigcup_{k=i}^j \mathcal{R}(p_k) = R' \\ 0 & \text{caso contrário} \end{cases}$$

Podemos assim, gulosamente computar os estados verificando se a restrição de cobertura é satisfeita para R , transformando a recorrência na seguinte equação:

$$(4) \quad dp(i, j) = \begin{cases} 1 & \text{se } \bigcup_{k=i}^j \mathcal{R}(p_k) = R \\ 0 & \text{caso contrário} \end{cases}$$

Podendo assim computar todos os estados polinomialmente. Porém, alguns dos estados computados são desnecessários, verificando que:

$$(5) \quad dp(i, j) = 1 \Rightarrow dp(a, b) = 1 \forall a \leq i, b \geq j$$

$$(6) \quad dp(i, j) = 0 \Rightarrow dp(a, b) = 0 \forall a \geq i, b \leq j$$

Uma vez que a operação de união é monotônica. Dessa forma não precisamos computar diversos estados baseando nos valores de outros estados. Descrevemos o nosso algoritmo guloso da seguinte forma: Seja $dp(i, j) = 0$ uma solução inicial. Incremente j de b tal que $dp(i, j+b) = 1$. Sabemos agora que não é necessário calcular os estados $dp(i, j')$ os quais $j' > j+b$ pela equação 5. Então, incremente i de a tal que $dp(i+a, j+b) = 0$. Sabemos neste momento que também não é necessário calcular os estados de $dp(i', j+b)$ com $i' > i+a$ pois pela equação 6 uma vez que sabemos o seus resultados. Repetimos o procedimento até que testemos todos os nós de G . O custo de execução e de memória é linear no número de vértices do grafo como no número de chaves, limitado em $O(|V| + |R|)$.

3. EXPERIMENTOS COMPUTACIONAIS

Nesta seção descrevemos os experimentos computacionais que forem realizados para verificar corretude do algoritmo e da análise assintótica descrita na seção anterior. Para tanto, foi criado um script em python `zikazeroaneldual.generator.py` que gera instâncias aleatórias para o melhor, pior e médio caso. Neste experimento,

geramos instâncias de tamanho $|V| = 5$ a $|V| = 25$ e $|R| = 5$ a $|R| = 20$ com distribuição aleatória de chaves por vértice. Cada instância foi executada 5 vezes, a fim de obter a média do tempo real gasto na execução dos algoritmos.

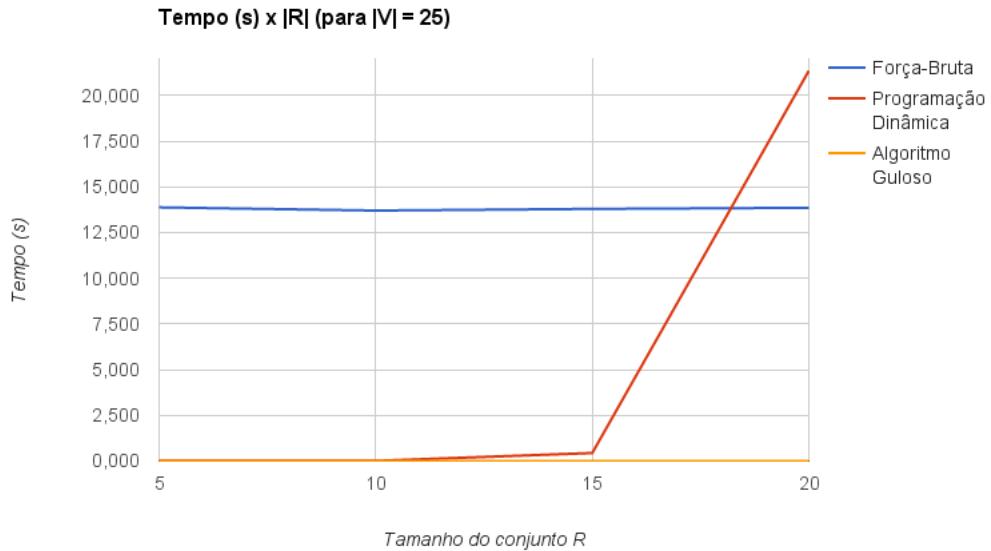


FIGURA 1. Gráfico de comparação de tempo de execução dos algoritmos para instâncias de tamanho $|V| = 25$

No gráfico apresentado na figura 1, variamos o tamanho do conjunto R para as instâncias de maior tamanho de V . O algoritmo de força-bruta se mantém constante como esperado, uma vez que seu custo de execução praticamente independe do tamanho de R . A programação dinâmica variou exponencialmente em relação a R , com podemos notar nos conjuntos de tamanho 15 para de tamanho 20. Por fim, o algoritmo guloso praticamente não variou, uma vez que seu custo é linear no tamanho da entrada, e como as entradas possuem tamanho muito pequeno, não é possível notar alguma variação expressiva no seu tempo de execução.

O gráfico apresentado em 2, fixamos o tamanho de R para a maior instância gerada, variando agora o tamanho do grafo G . Podemos notar neste gráfico que o algoritmo de força-bruta tem uma explosão exponencial entre grafos com 20 a 25 vértices, como esperado no seu custo assintótico. Já a curva do tempo de execução do algoritmo de programação dinâmica é notoriamente polinomial, sugerindo um custo maior do que o linear, como dito na seção anterior, se $|R|$ for fixo, o algoritmo de programação dinâmica tem custo polinomial de $O(|V|^2)$. Por fim, o algoritmo guloso também não se alterou conforme o gráfico anterior.

4. EXECUÇÃO

Essa seção trata dos aspectos envolvendo o algoritmo implementado em anexo ao trabalho. O código dos algoritmos encontram-se na pasta *source*. Para compilá-los e executá-los, basta digitar na linha de comando:

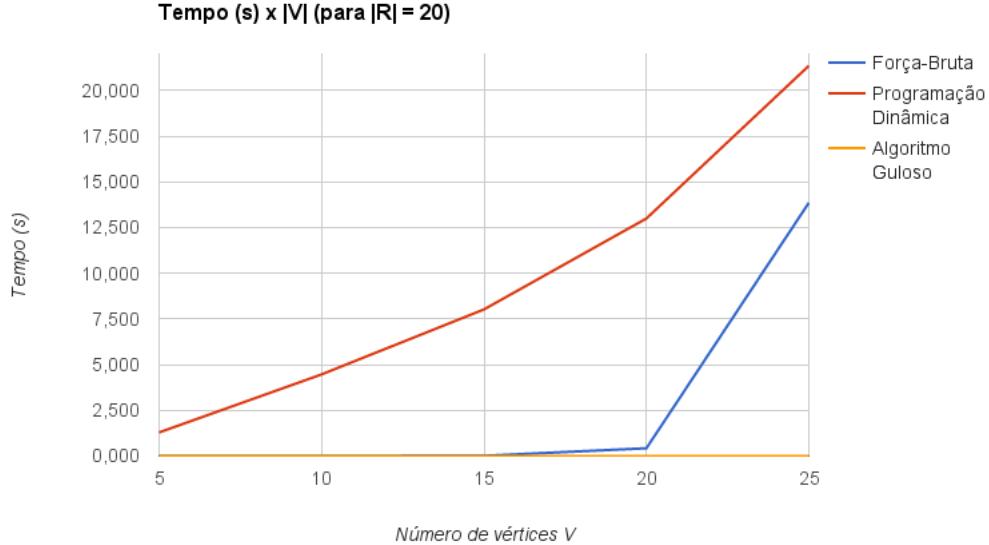


FIGURA 2. Gráfico de comparação de tempo de execução dos algoritmos para instâncias de tamanho $|R| = 20$

```
$> make brute_force
$> make dynamic_programming
$> make greedy
$> cd data
$> ./runner create
$> ./runner run
```

O grafo ótimo G^* encontrado pelos algoritmos é escrito nos arquivos *.sol*, enquanto o tempo de execução está descrito nos arquivos *.time*. Ao final, para limpar os dados, basta digitar na linha de comando:

```
$> ./runner clean
$> cd ..
$> make clean
```

REFERÊNCIAS

- [1] Korte, Bernhard; Vygen, Jeans, Combinatorial Optimization: Theory and Algorithms (5 ed.), Springer, ISBN 978-3-642-24487-2. 2012.
- [2] Bartle, Robert G. The elements of real analysis (second ed.). 1976.
- [3] Harary, Frank. Graph Theory, Addison-Wesley. 1969.
- [4] Donald Knuth. The Art of Computer Programming vol 1. Fundamental Algorithms, Third Edition. Addison-Wesley, 1997.
- [5] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. Introduction to Algorithms, Third Edition. The MIT Press. 2009.

Relatórios do Trabalho Prático de PAA:
ZikaZeroAnelDual

Daniel Kneipp de Sá Vieira

15 de junho de 2016

1 Exercício 1: Algoritmos

1.1 Busca por Força-bruta

Este algoritmo extrai subgrafos $G'(\mathbb{V}', \mathbb{A}')$ conexos do grafo original $G(\mathbb{V}, \mathbb{A})$ e testa se \mathbb{V}' atende todos os focos \mathbb{F} .

Os subgrafos são construídos começando do menor número de vértices possível que atenda a todos os focos até o grafo original. Como todos os vértices atendem a exatamente 2 focos, o menor número possível de vértices que atendem a todos os r focos é $\lceil \frac{r}{2} \rceil$.

O algoritmo para sua execução quando encontra o primeiro subgrafo que atende todos os focos.

Algoritmo 1: Força-bruta

Entrada: $G(V, A)$
Saída: V' de $G'(V', A')$ conexo com menor número de vértices que atende a todos os focos

```
1 início
2   |    $v_{min} := \lceil \frac{r}{2} \rceil;$ 
3   |   para  $i := v_{min} \dots |V|$  faz
4   |   |    $V' := [];$ 
5   |   |    $V'[1] := V[1];$ 
6   |   |   para  $j := 2 \dots i$  faz
7   |   |   |    $V'[|V'| + 1] := prox(V[|V'|]);$ 
8   |   |   para  $j := 1 \dots |V| - 1$  faz
9   |   |   |   se  $atendemTodosOsFocos(V')$  então
10  |   |   |   |    $ordena(V');$ 
11  |   |   |   |   retorna  $V'$ ;
12  |   |   |   senão
13  |   |   |   |    $rem(V'[1]);$ 
14  |   |   |   |    $V'[|V'| + 1] := prox(V[|V'|]);$ 
15  |   se  $atendemTodosOsFocos(V)$  então
16  |   |    $ordena(V);$ 
17  |   |   retorna  $V$ ;
18  |   senão
19  |   |   retorna  $\emptyset$ ;
```

1.2 Algoritmo Guloso

Este algoritmo utiliza o conceito de quantidade de conflitos para definir uma medida de comparação entre os voluntários. Se um dos focos acessados por um voluntário v_1 também é acessado por um outro voluntário v_2 , isso quer dizer que ambos os voluntários possuem ao menos um conflito.

Este conceito é utilizado porque os vértices com um menor número de conflitos acessam focos mais raros de serem acessados, sendo mais provável destes voluntários fazerem parte da solução ótima (bem como de qualquer outra). Mas note que este algoritmo não garante que encontrará a solução ótima.

Primeiramente é calculado o número de conflitos de todos os voluntários, após isso o voluntário com menor número de conflitos é escolhido. A partir deste primeiro voluntário é escolhido o adjacente a ele com menor número de conflitos. Iterativamente voluntários adjacentes à solução que está sendo construída que possuem o menor número de conflitos vão sendo adicionados até que uma solução factível seja entrada ou todos os voluntários sejam selecionados para fazerem parte da solução.

Algoritmo 2: Algoritmo Guloso

Entrada: $G(V, A)$
Saída: V' de $G'(V', A')$ conexo com menor número de vértices que atende a todos os focos

```
1 início
2   para  $v_1 \in V$  faca
3     para  $v_2 \in V \setminus \{v_1\}$  faca
4       se  $\mathcal{R}(v_1) \cap \mathcal{R}(v_2) \neq \emptyset$  então
5          $conf[v_1] := conf[v_1] + |\mathcal{R}(v_1) \cap \mathcal{R}(v_2)|;$ 
6
7        $sol := \{min(conf)\};$ 
8       se  $|\mathcal{R}(sol)| = r$  então
9         retorna  $sol;$ 
10      enquanto  $|sol| \leq |V|$  faca
11         $sol \cup$  adjacente de  $sol$  com menos conflitos;
12        se  $|\mathcal{R}(sol)| = r$  então
13          retorna  $ordena(sol);$ 
14
15 retorna  $\emptyset;$ 
```

1.3 Programação Dinâmica

Este algoritmo utiliza uma estrutura auxiliar $aux[|V|][r]$ semelhante a uma matriz mas cada linha pode possuir um número de colunas distintos uns dos outros.

Se interpreta essa estrutura da seguinte forma: a linha 0 por exemplo representa a quantidade de focos alcançados pela solução que possui como primeiro vértice o 0.

Para que essa interpretação pudesse ser feita o grafo do problema foi convertido para uma versão com grafo direcionado de tal forma que o sucessor de todo vértice não é seu antecessor. Isso foi feito para que não houvesse ambiguidade sobre qual o próximo vértice da solução representada por aux já que todos os vértices possuem 2 adjacentes no problema original. o número de vértice que compõe a solução é indicada pelo número de vezes que tal solução (ou linha) foi alterada.

Note que essa transformação não altera a complexidade do problema já todas as soluções do problema original continuam existindo nesta.

Essa estrutura possui $|V|$ linhas e caso o problema possua solução, a maior linha (que representará a solução) terá r colunas.

Sendo $anel$ o arranjo que representa o grafo transformado, a estrutura aux é preenchida da seguinte forma:

$$aux[i] = \begin{cases} \mathcal{R}(anel[i : v_{min}]), & \text{se } aux[i] = \emptyset \\ aux[i] \cup \mathcal{R}(anel[i + k]), & \text{caso contrário} \end{cases} \quad (1)$$

com k representando a quantidade de vértices usados em uma determinada

iteração do algoritmo. $anel[i : v_{min}]$ representa o subconjunto conexo dos vértices do anel começando com o vértice $anel[i]$ tendo v_{min} vértices.

O algoritmo para quando $|aux[i]| = r$.

Algoritmo 3: Programação Dinâmica

Entrada: $G(V, A)$

Saída: V' de $G'(V', A')$ conexo com menor número de vértices que atende a todos os focos

```

1 início
2    $v_{min} := \lceil \frac{r}{2} \rceil;$ 
3    $anel := obterAnel(G);$ 
4   para  $i := 1 \cdots |V|$  faz
5      $aux[i] := \mathcal{R}(anel[i : v_{min}]);$ 
6     se  $|aux[i]| = r$  então
7       retorna  $ordena(anel[i : v_{min}]);$ 
8   para  $i := v_{min} + 1 \cdots |V| - 1$  faz
9      $ii := i;$ 
10    para  $j := 1 \cdots |V|$  faz
11       $aux[j] := aux[j] \cup \mathcal{R}(anel[ii]);$ 
12      se  $|aux[j]| = r$  então
13        retorna  $ordena(anel[j : ii]);$ 
14       $ii := ii + 1;$ 
15    se  $\mathcal{R}(V) = r$  então
16      retorna  $ordena(V);$ 
17    senão
18      retorna  $\emptyset;$ 

```

Algoritmo 4: *obterAnel*

Entrada: $G(\mathbb{V}, \mathbb{A})$
Saída: $anel$

```

1 início
2   |    $anel[1] := V[1];$ 
3   |    $anel[2] := adjacentes(V[1])[1];$ 
4   |   para  $i := 3 \dots |V|$  faz
5   |     |    $adjs = adjacentes(anel[i - 1]);$ 
6   |     |   se  $adjs[1] \neq anel[i - 2]$  então
7   |     |     |    $anel[i] := adjs[1];$ 
8   |     |   senão
9   |     |     |    $anel[i] := adjs[2];$ 
10  |   para  $i := 1 \dots |V| - 1$  faz
11  |     |    $anel[|anel| + 1] = anel[i];$ 
12  | retorna  $anel;$ 

```

2 Exercício 2: Análise de Complexidade

2.1 Busca por Força-bruta

2.1.1 Análise temporal

A função que verifica se todos os focos são atendidos (na linha 9) possui um custo computacional de $O(2|\mathbb{V}'|) = O(|\mathbb{V}'|)$ já que se computa todos os focos atendidos pelos vértices e depois é comparado o número de focos distintos atendidos com o número total de focos existentes.

O número de vezes que esta função é executada no pior caso (quando a solução é todos os vértices ou não há solução) é:

$$\begin{aligned}
 & (|\mathbb{V}| - 1) \times (|\mathbb{V}| - \lceil \frac{r}{2} \rceil) + 1 \\
 & = |V|^2 + \lceil \frac{r}{2} \rceil - \lceil \frac{r}{2} \rceil |V| - |V| + 1 \\
 & = O(|\mathbb{V}|^2)
 \end{aligned} \tag{2}$$

Portanto, a complexidade temporal deste algoritmo é $O(|\mathbb{V}|^2) \times O(|\mathbb{V}'|) = O(|\mathbb{V}|^3)$.

A complexidade da ordenação $O(\mathbb{V}' \log_2(\mathbb{V}'))$ não é levada em conta na dedução acima pelo fato de não impactar significativamente na complexidade do algoritmo.

2.2 Análise espacial

Para se armazenar o grafo $G(\mathbb{V}, \mathbb{A})$ são utilizadas $2|\mathbb{V}|$ espaços de memória para $|\mathbb{V}| > 3$ utilizando listas de adjacência, com isso resultando em $O(|\mathbb{V}|)$ para armazenar o grafo G .

Armazenar os focos alcançados por cada voluntário custa $|\mathbb{V}|r$ caso todos os voluntários acessem todos os focos.

Na decorrência da execução do algoritmo se armazena além do grafo G a solução corrente que está sendo testada, que possui uma complexidade espacial de $O(|\mathbb{V}'|) = O(|\mathbb{V}|)$. Portanto a complexidade espacial deste algoritmo é $O(|\mathbb{V}| + |\mathbb{V}|r)$

2.3 Algoritmo Guloso

2.3.1 Análise temporal

O procedimento para calcular o número de conflitos custa $O(|\mathbb{V}|^2)$ já que se tem um número de focos constante e é uma operação de todos os vértices com todos. Já o procedimento para encontrar o vértice com menos conflitos custa $O(|\mathbb{V}|)$ em uma estrutura não ordenada.

Por fim, a etapa de construção da solução já com o primeiro vértice inserido possui custo constante para cada vértice inserido na solução. No pior caso, ou seja, quando todos os vértices fazem parte da solução, esta etapa possui custo $O(|\mathbb{V}|)$. A ordenação final gera um custo de $O(|\mathbb{V}| \log_2(|\mathbb{V}|))$.

Isso implica que a complexidade temporal deste algoritmo pode ser expresso em $O(|\mathbb{V}|^2)$.

2.4 Análise espacial

O algoritmo armazena o grafo $G(\mathbb{V}, \mathbb{A})$, os focos alcançados pelos vértices, a solução que está sendo construída e a quantidade de conflitos q cada vértice possui.

O grafo e os focos consomem juntos $O(|\mathbb{V}| + |\mathbb{V}|r) = O(|\mathbb{V}|r)$, como já foi discutido anteriormente. A solução corrente e a lista de número de conflitos dos vértices possuem uma complexidade espacial juntas de $O(2|\mathbb{V}|) = O(|\mathbb{V}|)$. Portanto o algoritmo guloso proposto possui uma complexidade espacial de $O(|\mathbb{V}| + |\mathbb{V}|r)$.

2.5 Programação Dinâmica

2.5.1 Análise temporal

Analizando o Algoritmo 4 (*obterAnel*) este possui duas estruturas de repetição que executam $O(|V|)$ vezes, logo, a complexidade deste algoritmo é $O(|V|)$.

As instruções das linhas 5 e 6 executam $O(|V|)$. Já que a linha 5 possui um custo de $O(|V|)$ isso resulta em $O(|V|^2)$ até a linha 8.

As linhas 11 e 12 são as mais executadas das estruturas de repetição que contemplam da linha 8 a 14. O número de vezes que estas são executadas é:

$$\begin{aligned}
& ((|V| - 1) - (v_{min} + 1) + 1) \times |V| \\
& = (|V| - 1)|V| - (v_{min} + 1)|V| + |V| \\
& = (|V|^2 - |V|) - (v_{min}|V| + |V|) + |V| \\
& = (|V|^2 - |V|) - (\lceil \frac{r}{2} \rceil |V| + |V|) + |V| \\
& = |V|^2 - |V| - \lceil \frac{r}{2} \rceil |V| - |V| + |V| \\
& = |V|^2 - \lceil \frac{r}{2} \rceil |V| - |V| \\
& = O(|V|^2)
\end{aligned} \tag{3}$$

Logo, a estrutura de repetição da linha 8 possui complexidade no pior caso de $O(|V|^2)$ com a união de dois conjuntos (linha 11) sendo feita em $O(1)$ com o número de focos fixo (igual a 2 neste caso).

Para se verificar se todos os vértices atendem a todos os focos (linha 15) basta adicionar os focos do vértice faltante em qualquer linha de *aux* após ter sido processada no *loop* da linha 8 e comparar com o tamanho do conjunto de focos original. Isso pode ser feito em $O(1)$.

A ordenação é feita apenas umas vez antes de se retornar o resultado (caso exista). Isso custa $O(|V| \log_2(|V|))$.

Portanto, a complexidade geral deste algoritmo é $O(|V|^2) + O(|V|^2) + (|V| \log_2(|V|)) = O(|V|^2)$.

2.6 Análise espacial

A estrutura auxiliar *aux* possui $|V|$ linhas a maior linha possui r colunas, logo esta estrutura gera uma complexidade espacial de $O(|V|r)$. *anel* possui um tamanho de $2|V| - 1 = O(|V|)$. O grafo $G(\mathbb{V}, \mathbb{A})$ que possui uma complexidade espacial de $O(|V|)$ como já foi analisado anteriormente. E por fim a estrutura que armazena os focos alcançáveis pelos voluntários custa em armazenamento $O(|V|r)$ no pior caso.

Com isso tem-se que a complexidade espacial geral da abordagem com programação dinâmica é $O(|V|r) + O(|V|r) + O(|V|) + O(|V|) = O(|V|r + |V|)$.

3 Exercício 4 e 5: Testes e Comparações

Os testes foram executados utilizando instâncias geradas automaticamente com número de voluntários variando de 10 a 1000 número de focos variando de 5 a 250. Nestas instâncias não há garantia de existência de solução.

A figura 1 mostra os tempos de execução obtidos do algoritmo de força bruta. É possível notar o crescimento acentuado do tempo gasto por conta da variação do tamanho da entrada mas ao mesmo tempo algumas instâncias possuem um

tempo de execução ligeiramente menor mesmo sendo relativamente maiores (é o caso da instância com 1000 voluntários e 188 focos, por exemplo). Isso deve-se ao fato de que o algoritmo para imediatamente após ter encontrado uma solução.

Note também que o algoritmo não possui uma piora significativa quando somente o número de focos é elevado já que o conjunto de soluções inciais que o algoritmo gera leva em conta o número mínimo de voluntários que uma solução deve ter para ser viável, eliminando construção e teste de soluções que com certeza não serão factíveis por conta do acréscimo do número de focos.

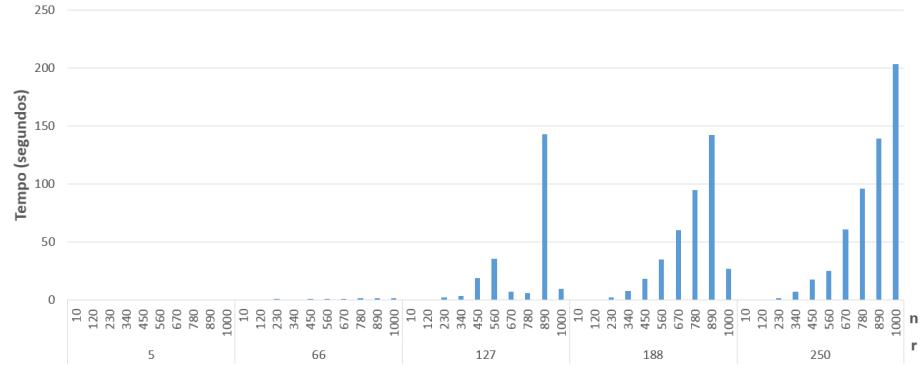


Figura 1: Tempos de execução do algoritmo de força bruta.

A figura 2 apresenta um comparativo entre o algoritmo baseado em programação dinâmica e o algoritmo guloso. Já que o algoritmo guloso não possui a poda de soluções de tamanho mínimo que o algoritmo de força bruta e dinâmico possuem, e também pelo fato de que sua complexidade temporal é atribuída por um procedimento que é executado completamente de qualquer forma, seu tempo de execução cresce em função do tamanho da entrada de forma muito mais padronizada que os outros.

Uma observações interessante é que o guloso não altera seu tempo de execução significativamente quando a quantidade de focos aumenta. Como foi previsto na análise de complexidade a etapa que consome mais processamento no algoritmo guloso tem seu custo determinado em função de $|V|$ e por conta disso ele executa em tempos muito próximos quando varia apenas o número de focos.

E por fim, é demonstrado que para instâncias relativamente grandes o algoritmo guloso supera em tempo de execução o algoritmo baseado em programação dinâmica, pelo custo de não achar a solução ótima.

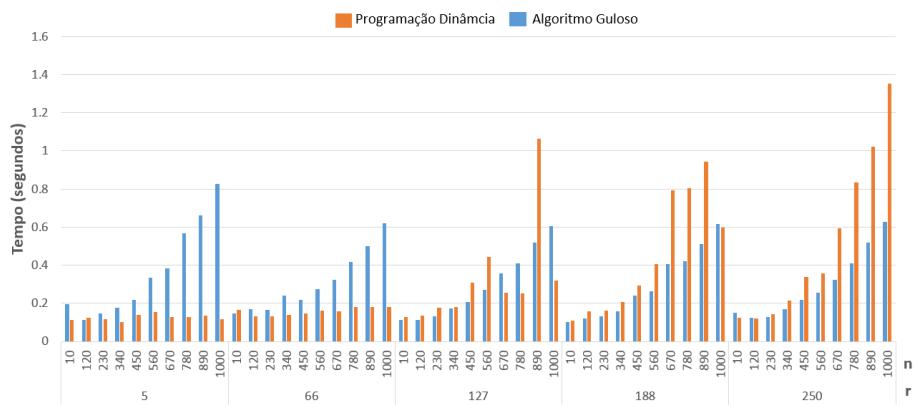


Figura 2: Comparativo de tempos de execução do algoritmo guloso e do algoritmo baseado em programação dinâmica.

Trabalho Prático: ZikaZeroAnelDual.

Samuel Moreira Abreu Araújo¹

¹Universidade Federal Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

samuelcco@gmail.com

Pré-textual: Para melhor entendimento, algumas nomenclaturas devem ser explicadas.

- n ou $|V|$, número total de vértices (voluntários).
- m ou $|E|$, número total de enlaces (relações).
- F conjunto de focos.
- E conjunto de arestas (relações).
- V conjunto de vértices (voluntários).
- f_i determinado foco $i \in F$.
- v_i determinado vértice $i \in V$.
- $m_{i,j}$ determinado enlace $(i, j) \in E$.
- n_f , número total de focos.
- f_i , número de focos acessados pelo vértice v_i .

1. Exercício - Proposta dos algoritmos

1.1. Busca Força-bruta

A busca exaustiva é um algoritmo simples, de uso abrangente e com custo computacional alto, neste caso em particular, de ordem polinomial. 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 averigua todo o espaço de soluções, por isso é garantido que ele sempre vai encontrar o ótimo (caso existir solução viável).

O desafio aqui foi gerar um algoritmo de Força-bruta (ou busca exaustiva), de complexidade polinomial em relação aos parâmetros de entrada, no caso o número de vértices (voluntários n), o número de enlaces (laços de amizade m) e o número de focos de cada vértice (n_f).

Por definição temos que o grafo de entrada $G(V, A)$ sempre vai ser do tipo anel, onde cada vértice conecta-se exatamente a dois outros vértices, ou seja sempre teremos um ciclo de tamanho par ou ímpar. Por esta definição também pode-se concluir que sempre $|V| = |A|$ ou seja $m = n$. Como pode ser observado na figura 1.

Partindo dessa premissa podemos diminuir o espaço de soluções, que antes era de exponencial da ordem de 2^n para uma complexidade da ordem polinomial n^2 , visto que o grafo é não orientado, exemplo pode ser visto na figura 1(a).

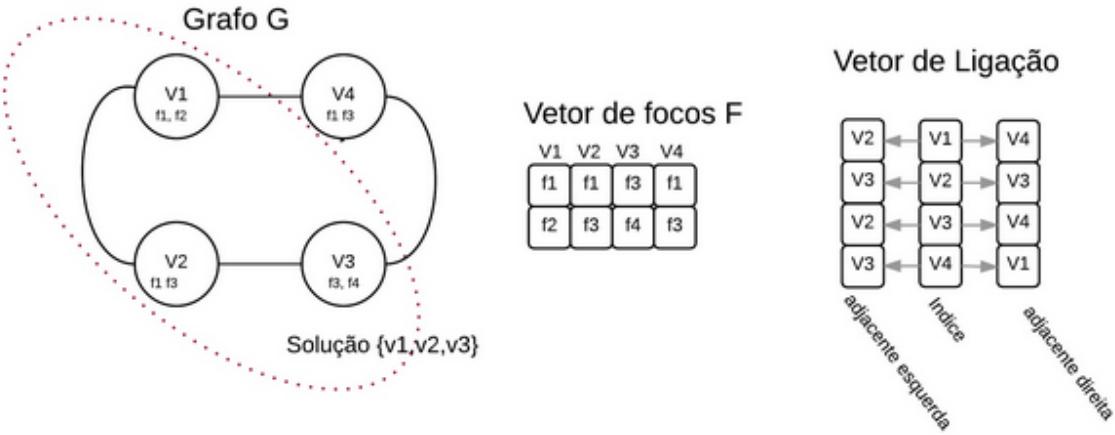


Figura 1. (a) Grafo Anel e solução Ótima,(b) Vetor de cobertura de focos e (c) Vetor de ligação a sucessores e predecessores.

O algoritmo de Força-bruta possui uma etapa de achar todos os caminhos, baseado em um método de caminhamento em grafos guloso, caminhando sempre para um sentido, no nosso caso a direita, até encontrar a própria origem. Gerando todas as sequencias de caminhamento viáveis do grafo G , e testando uma a uma. Considerando que cada caminho de v_i até v_j é representado em $V' \in G'$, cada caminho pode ter o tamanho 1, se contiver apenas um vértice, até n , se contiver todos os vértices.

Como no exemplo mostrado na figura 1(a), pelo fato de o grafo de entrada G ser anel, reduz muito o espaço de soluções, continuando no exemplo 1(a), admitindo-se o vértice v_1 com inicial, pode ser ter os seguintes grafos induzidos e conexos $G'[k]$:

1. $V \in G'[1] = \{v_1\}$
2. $V \in G'[2] = \{v_1, v_2\}$
3. $V \in G'[3] = \{v_1, v_2, v_3\}$
4. $V \in G'[4] = \{v_1, v_2, v_3, v_4\}$

No exemplo acima, o número de subgrafos induzidos partindo de v_1 em G é 4, que é justamente $|V|$, ou o valor de n . Mas nosso objetivo é achar todos os caminhos possíveis, logo essa sequencia deve ser iniciada de cada $v_i \in V$, deste modo teremos n^2 combinações diferentes de caminhos. Ainda por sequencia devemos ter em mente que os caminhos podem ser achados no sentido contrário, mas pelo fato do grafo por definição não ser orientado, podemos dizer que os subgrafos induzidos $G' = \{v_1, v_2, v_3, v_4\}$ e $G' = \{v_4, v_3, v_2, v_1\}$ são exatamente iguais porque não importa a sequencia que ele seja percorrido, desde que não de altere a ordem de caminhamento.

Os caminhos encontrados para o exemplo descrito na figura 1, podem ser melhor entendidos com a visualização da árvore de espaço de busca de caminhos viáveis gerada pelo grafo anel G , de acordo com a figura 2.

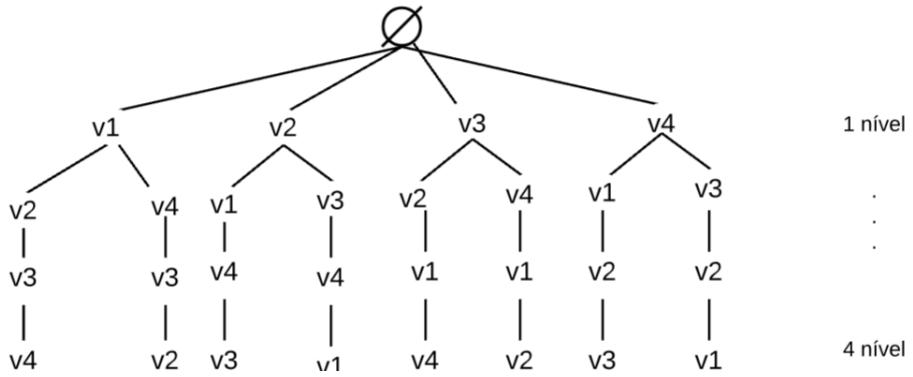


Figura 2. Árvore de soluções gerada pelo exemplo da figura 1.

Um algoritmo inocente contabilizaria duas vezes o caminho na árvore, uma vez caminhando para esquerda e outra para a direita, o que gera a saída de $2n^2$ caminhos, mas como o grafo é não dirigido, tal solução acaba sendo redundante, pois como mencionado $G' = \{v_1, v_2, v_3, v_4\}$ e $G' = \{v_4, v_3, v_2, v_1\}$ são iguais. Assim a arvore de solução mostrada na figura 2 pode ser simplificada ainda pela metade dos ramos e o número de soluções é n^2 .

O pseudo algoritmo de Força-bruta proposto, é apresentado no pseudocódigo algoritmo1.

Algoritmo 1 Força-bruta (G)

```

1: flag ← 0
2:  $G^* \leftarrow \infty$ 
3: for  $i \leftarrow 1$  to  $n^2$  do
4:    $G' \leftarrow$  Grafo induzido e conexo  $G'[i]$ 
5:   if VerificaFocos( $G' == \text{TRUE}$ ) then
6:     if  $G'.tam == G^*.tam$  and  $\text{sum}G'.sum \leq G^*.sum$  then
7:        $G^* \leftarrow G'$ 
8:       flag ← 1
9:     else if  $G'.tam < G^*.tam$  then
10:       $G^* \leftarrow G'$ 
11:      flag ← 1
12:    end if
13:  end if
14:   $i++$ 
15: end for
16: if flag == 1 then
17:   imprime ( $G^*$ )
18:   return TRUE
19: end if
20: return FALSE

```

Na linha 4 do algoritmo 1, existe um algoritmo de caminhamento guloso implícito que monta uma estrutura e gera cada caminho existente em G , sempre caminhando para

o próximo v_i adjacente a v_j , através da estrutura apresentada na figura 1(c). De posse de cada caminho, o mesmo é verificado e ente as linhas 5 e 14 do algoritmo 1, após a verificação o mesmo é sobreescrito pelo próximo caminho a ser testado.

De posse de cada caminho possível (linha 4, algoritmo 1) achados através do algoritmo incremental, basta testar cada caminho de forma a verificar se cobre todos os focos (linha 5, algoritmo 1). Não é necessário verificar se são conexos, pois pela propriedade da construção dos caminhos ele sempre gera soluções conexas. Ao final o melhor caminho testado (caso exista) é apresentado com a solução ótima para o problema (s^*) (linha 17, algoritmo 1).

Como melhor solução o algoritmo adota a que usa menos componentes (vértices), em caso de empate opta-se pela solução que apresentar o menor somatório dos índices dos vértices. Caso permaneça o empate no somatório dos índices, o algoritmo opta pelo que foi achado primeiro. Ao final da execução não é necessário ordenar a solução, pois construtivamente ela foi gerada de forma crescente.

1.2. Programação Dinâmica

Uma característica do problema abordado, se resolvido por força bruta, é a presença de cálculos redundantes, ou seja de ações que são avaliadas repetidas vezes. Tal fenômeno pode ser melhor observado na figura 3, onde independente do vértice que a solução começa a ser construída, sempre haverá a presença de cálculos redundantes. No exemplo demonstrado, baseado na figura 1(a), alguns casos de recálculo de subproblemas são demarcados de vermelho.

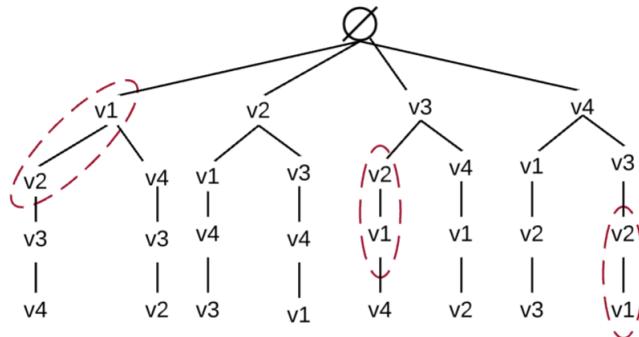


Figura 3. Árvore de soluções gerada pelo exemplo da figura 1(a), em vermelho o exemplo de alguns componentes que são calculados repetidas vezes.

O algoritmo proposto então faz uso de memória para reduzir o cálculo das ações repetidas, de tal forma a otimizar o desempenho quanto ao processamento. Mas em contra partida acaba gerando um consumo de memória para armazenar à as soluções anteriores.

Como estrutura de armazenamento foi usado uma matriz $n \times n$ onde os cálculos parciais serão armazenados e aproveitados em cálculos futuros. Matriz denominada $OPT[i, j]$, baseado no exemplo mostrado na figura 1(a), os cálculos gerados podem ser mostrados na tabela 1 e na equação de recorrência que segue.

Tabela 1. Matriz OPT $n \times n$, para exemplo apresentado na figura 1(a)

	1	2	3	4	(vértices)
1	(1,2)	(1,3)	(3,4)	(1,3)	
2	(1,2,3)	(1,3,2)	(3,4,1)	(1,3,4)	
3	(1,2,3,4)	(1,3,2)	(3,4,1,2)	(1,3,4)	
4	(1,2,3,4)	(1,3,2,4)	(3,4,1,2)	(1,3,4,2)	
	(iterações)				

- As colunas, representam os f_i focos que determinado vértice v_i acessa.
- As linhas, representam as i iterações, onde a cada iteração i , os f_i focos que determinado vértice v_i acessa são repassados ao v_{i+1} e unidos com os que ele já cobre.
- Na primeira iteração, na linha 1, são inseridos os focos f_i cobertos originalmente pelo voluntário v_i .
- Nas linhas seguintes, a cada nova iteração, a posição $\text{OPT}[i, j]$ recebe a união da posição acima dela $\text{OPT}[i - 1, j]$ com a da posição acima a esquerda dela $\text{OPT}[i - 1, j - 1]$. No caso de se estar no extremo da tabela e não existir $[j - 1]$, pega-se o valor da extrema direita da linha acima na tabela, formando uma sequência. Pode ser imaginado como um vetor, onde o final de cada linha é seguido do começo da linha a baixo.
- Os cálculos são feitos linha a linha, até se chegar a uma solução.

Abaixo segue a equação de recorrência a ser seguida pelo algoritmo.

$$\underbrace{\text{OPT}[i, j]}_{0 < i < n \text{ e } 0 < j < n} = \begin{cases} f_j \in v_j, & \text{se } i = 0. \\ \text{OPT}[i - 1, j] \cup \text{OPT}[i - 1, j - 1], & \text{se } j \geq 1. \\ \text{OPT}[i - 1, j] \cup \text{OPT}[i - 1, |V|], & \text{se } i > 0 \text{ e } j = 0. \end{cases}$$

Onde f_j é o conjunto de focos originais coberto por pelo voluntário j .

O algoritmo em si é trivial, e funciona como se cada voluntário da rede (Grafo anel G) falasse para seu sucessor que focos ele cobre, com base nessa informação, o sucessor avalia se consegue cobrir todos os focos, caso não repassa toda essa informação a frente. Dessa maneira, propaga-se o alcance de focos de cada voluntário a cada iteração. No momento em que um voluntário possuir todas as coberturas de focos ele observa quem são seus predecessores que vão ajudá-lo a combater os focos.

A tabela 1, apresenta a execução até o final do algoritmo de Programação Dinâmica para o exemplo exibido na figura 1(a). As linhas da tabela representam as iterações feitas e as colunas os vértices calculados. Repare que para a execução correta do algoritmo, cada linha deve conter os vértices na ordem em que são adjacentes no garfo G . Ou seja, no exemplo, 1 é adjacente a 2, 2 é adjacente a 3, 3 é adjacente a 4 e por fim 4 é adjacente a 1, formando o anel.

Repare que na execução da tabela 1 o algoritmo poderia encerrar os cálculos no fim da execução da iteração 3. Onde as posições $\text{OPT}[3, 1]$ e $\text{OPT}[3, 3]$, possuem as primeiras soluções viáveis encontradas.

Ao fim de cada iteração o algoritmo verifica se cada célula da iteração analisada possui uma solução, caso a solução ocorra em duas ou mais células, o que é pertinente, similarmente ao algoritmo de Força-bruta, o algoritmo de Programação Dinâmica opta por exibir a solução com menor somatório dos índices dos vértices, caso ainda ocorra o empate, ele opta por mostrar a que foi primeiro encontrada.

A solução encontrada pode então ser construída pelos deslocamentos feitos na Tabela OPT , no exemplo da tabela 1, a solução foi achada nas células $OPT[3, 1]$ e $OPT[3, 3]$ na terceira iteração, no caso as duas células são comparadas então.

- $OPT[3, 1]$ iteração 3, logo solução possui os vértices $j = 1, j - 1 = 4$ e $j - 2 = 3$, quando $j - n < 0$, retorna a posição final da linha anterior, como ocorrido neste caso. Logo a solução é formada pelos voluntários $s = \{1, 4, 3\}$, e $sum(s) = 8$.
- $OPT[3, 3]$ iteração 3, logo solução possui os vértices $j = 3, j - 1 = 2$ e $j - 2 = 1$, $s = \{3, 2, 1\}$, e $sum(s) = 6$. Neste caso não foi preciso retornar a linha anterior.

A melhor solução então é $OPT[3, 3]$, pois possui somatório de índices igual 6;

O algoritmo de Programação Dinâmica pode ser melhor entendido pelo pseudo-código 2, mostrado a seguir:

Algoritmo 2 Programação Dinâmica (G)

```

1: for  $i = 0$  to  $n$  do
2:   for  $j = 0$  to  $n$  do
3:     if  $i == 0$  then
4:        $OPT[i, j] \leftarrow cobertura\_original(v_i)$ 
5:     else if ( $i > 0$ ) e ( $j == 0$ ) then
6:        $OPT[i, j] \leftarrow OPT[i - 1, j] \cup OPT[i, |V|]$ 
7:     else if  $j > 0$  then
8:        $OPT[i, j] \leftarrow OPT[i - 1, j] \cup OPT[i - 1, j - 1]$ 
9:     end if
10:    if ( $verificasolucao(OPT, i) == 1$ ) then
11:       $ordena(OPT, i)$ 
12:      return true
13:    end if
14:  end for
15:  return false
16: end for

```

Pelo algoritmo de Programação Dinâmica o problema pode ser resolvido em até n iterações. Caso se chegue a enésima iteração sem a resposta, o problema é dito inviável, pois cada posição vai possuir a cobertura máxima, logo existe um foco que não é coberto por nenhum voluntário.

O algoritmo proposto deve percorrer toda a tabela OPT , da esquerda para a direita, de cima para baixo até achar a linha da solução. A tabela possui dimensão máxima de $n \times n$. Cada união de células é feita em $O(f_j^2)$, pois deve-se ver os focos que a célula já cobre para não repetir a inserção de focos já cobertos. Para cada célula existe a verificação se ela cobre todos os focos que é feito em $O(f_j)$.

Ao final do algoritmo, na linha 11, caso exista solução viável, o algoritmo ordena a solução com base no algoritmo *Merge sort*, onde segundo [Cormen et al. 2009], possui complexidade temporal de $\Theta(n \log n)$ e de espaço de $\Theta(n)$.

1.3. Algoritmo Guloso

O algoritmo guloso para o problema do ZikaZeroAnelDual, explora a condição de entrada do problema. Que coloca que o grafo de entrada seja um grafo anel, onde cada vértice possui a possibilidade de se comunicar diretamente somente com dois outros vértices adjacentes a ele, formando um anel.

Visto que cada vértice possui apenas duas possibilidades de caminhar, para frente ou para trás, o algoritmo usa como critério de escolha gulosa sempre escolher o vértice v_i que está mais próximo localmente de tornar a solução ótima. Tal escolha gulosa é feita com uma função que calcula o quanto determinado vértice está próximo a tornar a solução corrente ótima globalmente. Em caso de haver empate entre as distâncias achadas entre cada vértice v_i , o algoritmo opta por selecionar o vértice que possui um menor somatório menor das distâncias. E assim insere v_i na solução S . Ao final das iterações o conjunto de ótimos locais irá gerar o ótimo global, devido a subestrutura ótima do problema.

A subestrutura ótima está nas características de que o grafo é anel e a solução deve ser conexa, onde qualquer subcaminho de um caminho que leve a solução ótima, também é um caminho ótimo localmente. Suponhamos que partindo de um vértice v_i qualquer, exista um vértice v' que leve ao ótimo mais rapidamente que os vértices $v_{(i-1)}$ ou $v_{(i+1)}$, o vértice v' deveria então fazer parte da solução, o que é um absurdo, pois como o grafo é anel, v_i só pode ter dois adjacentes que são $v_{(i-1)}$ e $v_{(i+1)}$, onde um dos dois deverá fazer parte da solução ótima global.

O algoritmo guloso pode ser melhor entendido pelo pseudocódigo 3, mostrado a seguir.

Algoritmo 3 Guloso (G)

```

1:  $S \leftarrow \emptyset$ 
2:  $Q = \{v_1, v_2, v_3, \dots, v_n\}$ 
3:  $F = \{f_1, f_2, f_3, \dots, f_n\}$ 
4:  $T \leftarrow Constroi_tabela_distancias(G, Q, F)$ 
5: while  $F \neq \emptyset$  do
6:    $v_i \leftarrow min\_dmax\_adj(T, S)$ 
7:    $S \leftarrow S \cup \{v_i\}$ 
8:    $Q \leftarrow Q - \{v_i\}$ 
9:    $F \leftarrow F - \{f(v_i)\}$ 
10:   $T \leftarrow Atualiza_tabela_distancias(G, Q, F)$ 
11: end while
```

Para calcular o vértice v_i a ser incluído na solução, o algoritmo monta uma tabela de distâncias de ordem $n \times n_f$, onde as linhas são os voluntários e as colunas os f focos. Cada célula contém a distância mínima do vértice v_i a cada foco f_j , função chamada de $d(v_i, f_j)$. A tabela é construída apenas uma vez, ao início do algoritmo, após basta atualizar cada coluna já coberta.

A função $d(v_i, f_j)$ implícito na construção da tabela de distâncias (algoritmo 2, linha 4), usa como método de caminhamento em grafos um modelo adaptado de busca em largura, que retorna a menor distância em saltos de cada voluntário $v_i \in V$ a cada foco $f_j \in F$. A função $sum(d_i)$ é usada no critério de desempate e calcula a soma de todas as mínimas distâncias $d(v_i, f_j)$ presentes para o vértice v_i .

Na etapa inicial, mostrada na Tabela 2, a tabela é preenchida com a distância de cada elemento até determinado foco (algoritmo 2, linha 4). A função $d(max)$ é a maior distância encontrada entre cada vértice e o foco $d(v_i, f_j)$. O vértice escolhido então é o que possui menor $d(max)$, em caso de empate o menor $sum(d_i)$ e caso o empate ainda persista, escolhe-se o que têm menor índice inicial, no caso mostrado na tabela 2, é escolhido o vértice v_2 .

A escolha do ótimo local é feita com base em uma matriz de cálculos apresentada na tabela 2 para o exemplo retirado da figura 1(a). Onde o vértice v_i a ser escolhido é aquele que possui uma proximidade maior em obter o ótimo local, e que seja adjacente a algum integrante do conjunto S visitados, afim de garantir a conexidade. Em caso de empates, é escolhido o de menor índice para ser inserido na solução.

Tabela 2. Algoritmo guloso, exemplo retirado da figura 1, iteração 1.

Vértice	Foco $d(v_i, f_i)$				$d(max)$	$sum(d_i)$
	$f1$	$f2$	$f3$	$f4$		
$v1$	0	0	1	2	2	5
$v2$	0	1	0	1	1	3
$v3$	1	2	0	0	2	5
$v4$	0	1	0	1	1	3

Iteração 1:

- Solução parcial $S = \emptyset$
- Candidatos $Q = \{v_1, v_2, v_3, v_4\}$
- Escolha Gulosa $v_i = v_2$
- Focos $F = \{f_1, f_2, f_3, f_4\}$

Na iteração seguinte, é executado a fase de redução da tabela (algoritmo 3, linha 10), onde os focos cobertos pelo vértice escolhido anteriormente são eliminados, e recalcula-se o $d(max)$ e $sum(d_i)$ de cada vértice com a tabela residual. No exemplo elimina-se f_1 e f_3 , que são cobertos por v_2 . Assim como mostrado na tabela 3.

Tabela 3. Algoritmo guloso, exemplo retirado da figura 1, iteração 2.

Vértice	Foco $d(v_i, f_i)$				$d(max)$	$sum(d_i)$
	$f1$	$f2$	$f3$	$f4$		
$v1$	-	0	-	2	2	4
$v2$	-	-	-	-	-	-
$v3$	-	2	-	0	2	4
$v4$	-	1	-	1	1	3

Iteração 2:

- Solução parcial $S = \{v_2\}$
- Candidatos $Q = \{v_1, v_3, v_4\}$
- Escolha Gulosa $v_i = v_1$
- Focos $F = \{f_2, f_4\}$

Nesta etapa escolhe-se o vértice adjacente a $v_i \in S$ que atendem o critério guloso, assim será escolhido v_1 , por possuir menor índice que v_3 . Repete-se o processo de redução da tabela, o que gera a tabela 4, mostrada na sequencia.

Tabela 4. Algoritmo guloso, exemplo retirado da figura 1, iteração 3.

Vértice	Foco $d(v_i, f_i)$				$d(\max)$	$\sum(d_i)$
	f1	f2	f3	f4		
v1	-	-	-	-	-	-
v2	-	-	-	-	-	-
v3	-	-	-	0	0	0
v4	-	-	-	1	1	2

Iteração 3:

- $S = \{v_1, v_2\}$
- $Q = \{v_3, v_4\}$
- Escolha Gulosa $v_i = v_3$
- Focos $F = \{f_4\}$

Escolhe-se mais uma vez os vértices adjacentes a $S = \{v_1, v_2\}$ que atendem o critério guloso, neste caso será escolhido v_3 , por possuir menor índice que v_4 . Repete-se o processo de redução da tabela.

Iteração 4:

- $S = \{v_1, v_2, v_3\}$
- $Q = \{v_4\}$
- Critério Guloso $v_i = f_{im}$
- Focos cobertos $F = \emptyset$

Percebe-se que ao final da terceira iteração, todos os focos estão cobertos (conjunto F de focos vazio), logo o algoritmo chega a seu critério de parada. Nesta etapa então a solução é dada por $s = \{v_1, v_2, v_3\}$.

Em resumo, o algoritmo usa o princípio de escolha gulosa que maximiza a proximidade que gera a solução de cobertura de focos baseado na tabela de distâncias.

Para montar a tabela de distâncias, para cada célula da tabela $n \times n_f$ é preciso verificar a distância para determinado foco f_i , o que é feito com uma busca em largura, onde segundo [Cormen et al. 2009], possui complexidade de tempo de $O(n + m)$ e de espaço de $O(n)$, pois é representada usando uma lista de adjacência.

Para atualizar a tabela de distâncias, é preciso visitar cada célula presente na tabela de distâncias, tal ação demanda o custo $O(nn_f)$. As inserções e remoções feitas nos conjuntos S, Q e F , são feitas em tempo constante $O(1)$ (vetores acessados pelo índice). A verificação se todos os focos estão cobertos é feita de maneira linear ao número de focos, $O(n_f)$.

2. Complexidades Temporal e Espacial

2.1. Busca por Força-bruta

2.1.1. Análise de tempo

- Geração de soluções (caminhos válidos), linha 4 do algoritmo 1. executa n^2 vezes. Sua complexidade é dada pelo número de saídas geradas, o que dá a característica polinomial do problema. Complexidade $O(n^2)$, onde para cada saída é executada uma vez o restante do código.
- Verificar focos, linha 5 do algoritmo 1, executada uma vez a cada entrada, faz a verificação para cada nó de uma possível solução. Verifica se a entrada acessa todos os focos $f_i \in F$. Complexidade no pior caso $O(nn_f)$.
- Copiar a solução, para cada solução melhorada, a solução é copiada para um vetor denominado S^* , feito em $O(n)$.
- Inicializar os parâmetros, vetor de focos é setado com 1 para todos os focos, sempre a cada nova solução testada. A cada foco achado vetor é setado na posição do foco f_i com 0. Executado em $O(n_f)$.

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 de Força-bruta executa em tempo polinomial da ordem de $O(n^2(nn_f + n + n_f)) = O(n^3n_f)$.

2.1.2. Análise de espaço

- Vetor de soluções G' , usado apenas para preservar os dados. Alocado somente uma vez no programa e sobreposto a cada iteração. Declarado como G' no pseudocódigo do algoritmo 1). Complexidade $O(n)$.
- Vetor $vFoco[][]$, estrutura usada para armazenar os focos existentes de cada vértice, (imagem 1(b)). Cada célula possui um *unsigned short int* que armazena o foco coberto por cada voluntário. Complexidade $O(2n)$, sabendo que cada voluntário possui dois focos.
- Vetor $f[]$, estrutura usada para verificar os focos já cobertos, possui o total dos focos, usado para verificação da solução. Complexidade $O(n_f)$.
- 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 algoritmo 1). O vértice é acessado pelo índice de alocação. Complexidade $O(n)$.

Ao final temos que o custo de espaço utilizado pelo algoritmo de Força-bruta é $O(4n + n_f)$.

2.2. Programação Dinâmica

2.2.1. Análise de tempo

As operações são feitas sobre a matriz OPT $n \times n$, são o que induzem a complexidade polinomial do algoritmo, dado por, $O(n^2 * custo_atualizacao)$.

Internamente dentro dos dois loops aninhados, para cada célula temos:

- Copiar cobertura original dos voluntários, no caso de $i = 0$, para todo f_i , o que é feito em $O(2)$, sabendo-se que cada voluntário acessa dois focos (algoritmo 2, linha 4).
- Copiar cobertura das posições anteriores a $\text{OPT}[i, j]$, para $i > 0$, o que é feito em $O(n_f^2)$, pois deve-se verificar para não copiar focos f_i redundantes (algoritmo 2, linha 6 e 8).
- Verificar a solução (algoritmo 2, linha 10), para cada $\text{celula}_{(i,j)}$ da linha_i , deve ser verificado se chegou-se a solução. Para cada célula, a verificação é feita em $O(n_f)$.

Ao final temos que para cada célula presente na tabela existe o custo de $O(2 + n_f^2 + n_f) = O(n_f^2)$, multiplicando para cada célula da tabela OPT, temos o custo de tempo final do algoritmo de Programação Dinâmica na ordem de $O(n^2 n_f^2)$.

2.2.2. Análise de espaço

- A tabela $\text{OPT}nxn$, consome o espaço máximo da ordem de $O = (n^2)$, onde cada célula contém um vetor que poderá armazenar todos os n_f focos existentes. Logo complexidade de espaço de $O(n^2 n_f)$.
- Vetor que armazena os focos de cada vértice, lido diretamente do arquivo, complexidade de espaço de $O(2n)$ pois cada vértice possui dois focos.
- Vetor de adjacência, armazena os vértices e seus adjacentes, acessado pelo índice, de posse da informação que cada vértice possui dois adjacentes, complexidade de tamanho da ordem de $O(2n)$, similar a figura 1(b).

Ao final da implementação, o algoritmo de Programação Dinâmica possui a complexidade de espaço total na ordem de $O(n^2 n_f + 4n)$

2.3. Algoritmo Guloso

2.3.1. Análise de tempo

Inicialmente o algoritmo:

- Preenche o vetor Q com todos os vértices do grafo, algoritmo 3, linha 2. Complexidade $O(n)$.
- Preenche o vetor F com todos os focos existentes, algoritmo 3, linha 3. Complexidade $O(n_f)$.

Em seguida, o algoritmo constrói uma tabela de distâncias, para consultar os vértices a serem gulosamente escolhidos, esta construção é feita somente uma vez, no algoritmo 3, linha 4. Como a tabela de ordem tamanho nxn_f e para cada célula da tabela é feita uma Busca em Largura, esta etapa tem a complexidade de $O(nn_f(n+m))$, sabendo que $m = n$, pode-se afirmar que o custo é $O(n^2 n_f)$.

Na linha 5 do algoritmo 3, existe um *loop* que retira um vértice v_i por vez, até todos os focos estarem cobertos. Como o número de focos pode ser de tamanho até $2n$, cada vértice v_i possui no máximo 2 focos, no pior caso, este *loop*, executa até n vezes. Para cada iteração temos os seguintes custos:

- Escolha Gulosa, linha 6 do algoritmo 3, verifica em $O(2n)$ o menor valor da coluna $d(\max)$ em caso de empate deve verificar a coluna $sum(d_i)$.
- Inserção em S , linha 7 do algoritmo 3, insere em tempo $O(1)$.
- Remoção em Q , linha 8 do algoritmo 3, remove em tempo $O(n)$.
- Remoção em F , linha 9 do algoritmo 3, remove em tempo $O(n_f)$.
- Atualiza a tabela de distâncias, linha 10 do algoritmo 3, sabendo que cada nó possui dois focos, deve-se zerar até duas colunas de tamanho n , logo $O(2n)$, deve-se recalcular os valores de $d(\max)$ e $d(sum)$, para isso deve-se visitar todas as linhas da tabela. Complexidade de $O(nn_f)$

Logo a complexidade temporal do algoritmo Guloso é da ordem de $O((nn_f(n + m)) + n(2n + 1 + n + n_f)) = O(n^2n_f)$

2.3.2. Análise de espaço

- Vetor Q , recebe cada voluntário, tamanho da ordem de $O(n)$.
- Vetor S , recebe cada voluntário que entra na solução, tamanho da ordem de $O(n)$.
- Vetor F , recebe cada foco existente, tamanho da ordem de $O(n_f)$.
- Tabela distâncias, no eixo X os voluntários e no eixo y os focos, de tamanho $O(nn_f)$.

Logo o consumo máximo de memória utilizado pelo algoritmo Guloso é da ordem de $O(2n + n_f + nn_f)$.

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

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

Exemplo de execução

```
$ ./executar-dinamico.sh exemplos/exemplo1.txt saída.txt
```

4. Testes computacionais

Os testes deste capítulo foram divididos em duas seções para serem melhor explicados. Na primeira seção são feitos testes de exemplos que demonstram a corretude experimental computacional dos métodos. No segundo são mostrados testes feitos para análise de carga e tempo.

A conectividade entre cada par de vértices foi fixada em um, onde o último vértice se conecta ao primeiro, para que sempre possamos ter como saída um grafo anel.

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.

4.1. Teste de corretude

Inicialmente todos os testes disponibilizados no fórum foram rodados, e suas saídas verificadas com sucesso. Em todos os casos de testes, in , $in1$, $in2$, $in3$, $in4$ e $in5$, o resultado ótimo foi encontrado em todas as variações do algoritmo, como o esperado.

Para os demais testes de corretude, foram geradas algumas instâncias aleatórias de acordo com uma distribuição de focos planar uniforme, que são plotadas com um *script* em *opengl* desenvolvido para este fim. Dessa maneira podemos verificar visualmente as soluções encontradas.

Na geração das visualizações, as coordenadas x, y de cada $vi \in V$ são geradas aleatoriamente, lembrando que esta não interfere na classificação do grafo como anel. A característica para o grafo ser anel é cada vértice possuir $d(vi) = 2$ onde $d(vi)$ é o grau de adjacência de cada vértice.

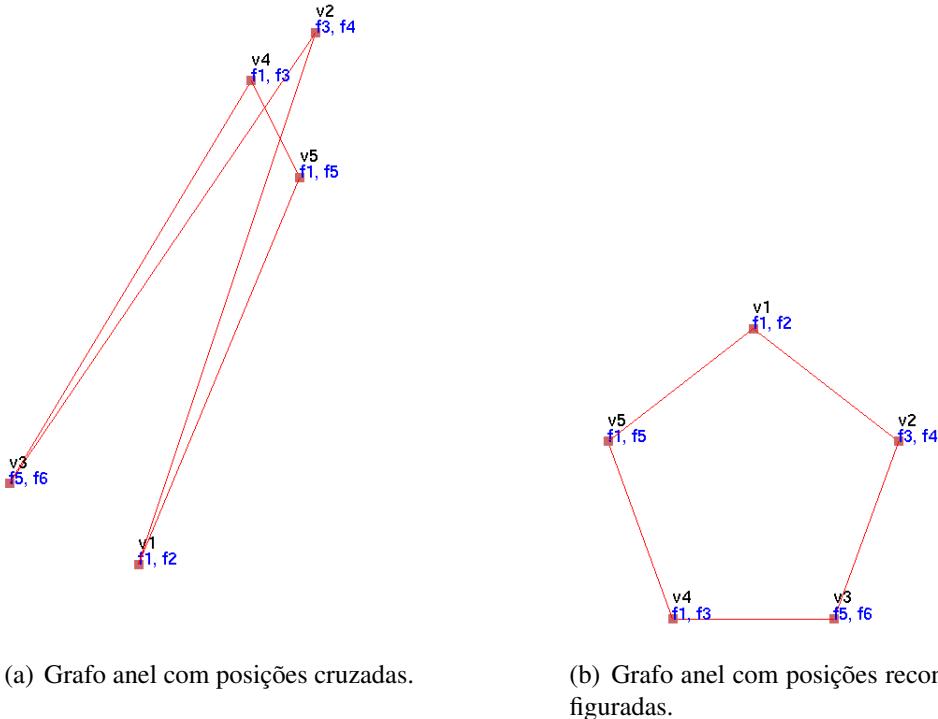


Figura 4. Exemplo de grafo anel com 5 vértices e 5 arestas e 6 focos diferentes.

Como podemos ver na figura 4(b), o grafo é anel pois atende as requisições para tal. A figura 4(a) é isomórfica a figura 4(b), logo também é anel.

No teste realizado com base na instância mostrada na figura 4, foram testados os três algoritmos. Onde os três apresentaram o mesmo resultado, sendo este ótimo, $S^* = \{v_1, v_2, v_3\}$.

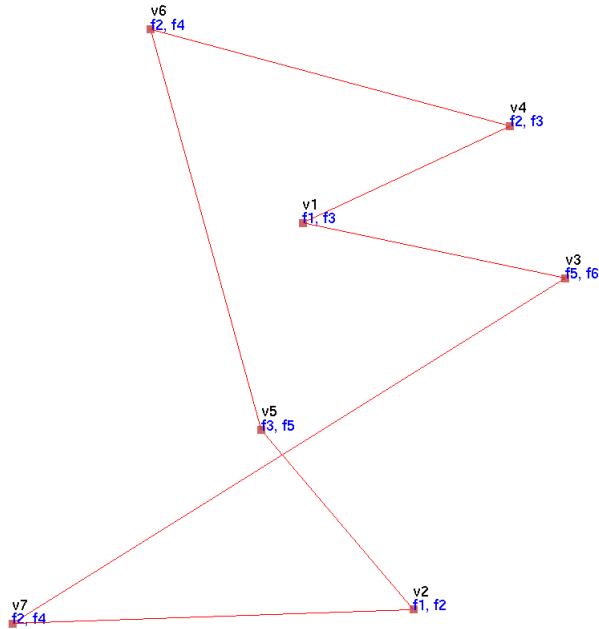


Figura 5. Grafo Anel, solução ótima $S^* = \{v_1, v_3, v_7\}$.

O teste mostrado na figura 5, apresentam um grafo que demonstra 7 voluntários (vértices), 7 relações de amizade (enlaces) entre v_i, v_j e o número total de focos diferentes igual a 6. Este exemplo é bom para mostrar o funcionamento do algoritmo mesmo quando os voluntários não estão em ordem em relação aos índices, ou seja nem sempre na relação (v_i, v_j) temos que $v_j = v_{i+1}$ ou $v_i = v_{j-1}$.

Nos testes realizados, na figura 5, todas as três implementações apresentaram o resultado ótimo, sendo, $S^* = \{v_1, v_3, v_7\}$.

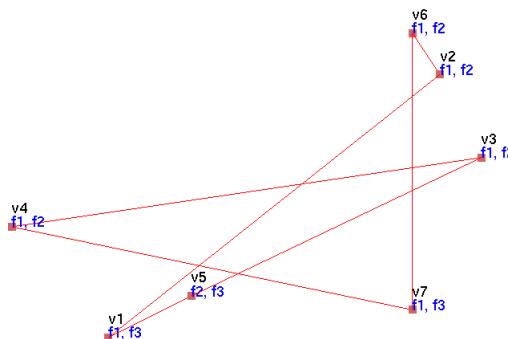


Figura 6. Grafo Anel, solução ótima $S^* = \{v_1, v_2\}$.

O grafo da figura 6, possui 7 vértices e 3 focos diferentes, todas as três implementações apresentaram o resultado ótimo, sendo, $S^* = \{v_1, v_2\}$.

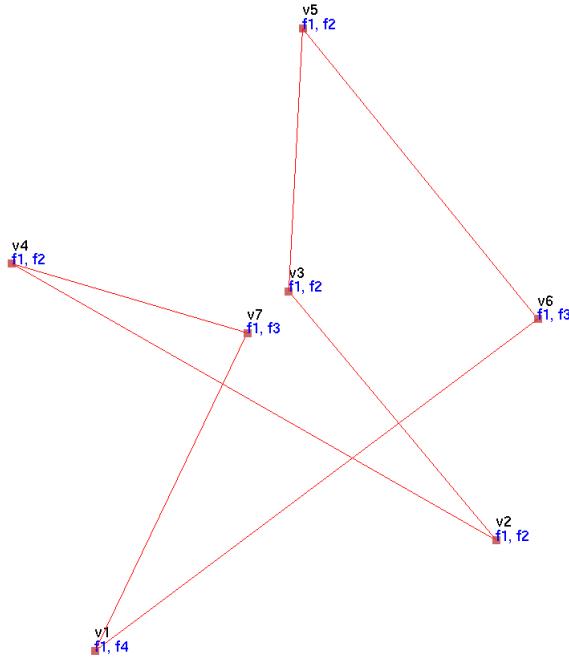


Figura 7. Grafo Anel, solução ótima $S^* = \{v_1, v_5, v_6\}$.

O grafo da figura 7, possui 7 vértices e 4 focos diferentes, todas as três implementações apresentaram o resultado ótimo, sendo, $S^* = \{v_1, v_5, v_6\}$. Repare que para este caso existem duas soluções ótimas e que os somatório dos índices das duas são iguais, onde:

- $S_1 = \{v_1, v_5, v_6\}$, onde $1 + 5 + 6 = 12$.
- $S_2 = \{v_1, v_4, v_7\}$, onde $1 + 4 + 7 = 12$.

Em casos de empates assim, todas as implementações optam por armazenar a solução encontrada primeiro, neste caso S_1 , afim de reduzir o número de comparações.

O grafo da figura 8, possui 100 vértices e 10 focos diferentes. Tal teste é importante pois verifica-se a corretude dos algoritmos para instâncias maiores.

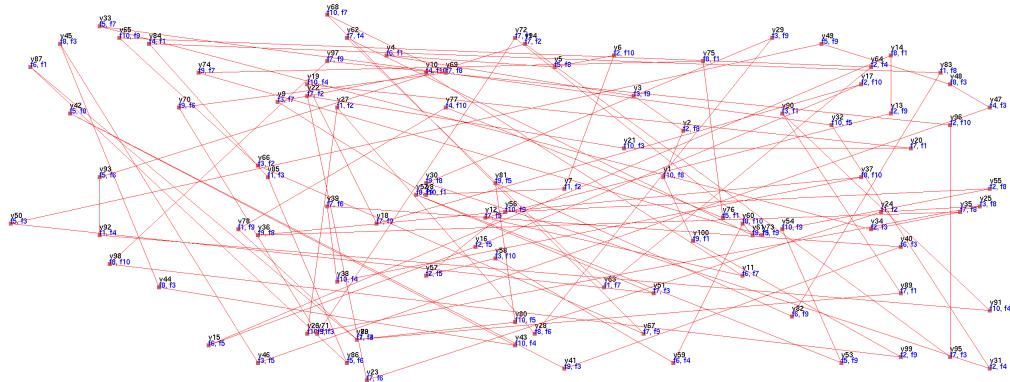


Figura 8. Grafo Anel, solução ótima $S^* = \{v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}\}$.

Para a instância mostrada na figura 8, todos os três algoritmos apresentaram o mesmo resultado, $S^* = \{v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}\}$.

Para testes maiores como o apresentado na figura 8, o resultado já é difícil de ser verificado visualmente, mas pela indução embasada na teoria da construção adotada nos algoritmos, e pelas três implementações exibirem resultados iguais, acredita-se na corretude dos resultados. Outros testes maiores foram executados e tiveram suas saídas comparadas, mas não foram plotados por inviabilidade de espaço, mas seus tempos de execução irão ser mostrados na seção 4.2.

4.2. Teste de desempenho

Para os testes apresentados nesta seção, as instâncias foram geradas com uma distribuição uniforme, variando em números de focos f e voluntários. Em todos os testes, foram geradas 10 instâncias aleatórias para cada cenário, onde os gráficos foram plotados com a média de tempo encontrada em cada caso, afim de gerar uma maior fidelidade nos dados. Tais testes são úteis para perceber os cenários em que determinados métodos são mais indicados.

Nos testes mostrados nas figuras 9, 10 e 11, o número de vértices variou de 100 a 750. Por tratar-se de um grafo anel o número de enlaces foi o mesmo, já o número de focos foi aumentado proporcionalmente a uma taxa de 10% sobre o número de vértices, ou seja, para 250 vértices temos 25 enlaces.

Analizando o algoritmo de Força-bruta pela figura 9, percebemos que o mesmo não altera seu comportamento quanto ao crescimento de nós e focos proporcionais. Tal comportamento indica que a complexidade é gerada em função do número de possíveis soluções a serem testadas, que no caso é $O(n^2)$, e não do tempo de verificação da solução, que é $O(nn_f)$ como visto na seção de 2.3.

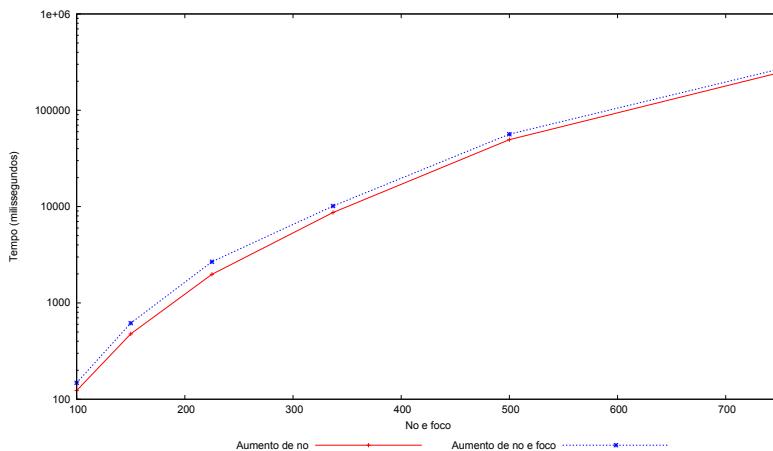


Figura 9. Algoritmo de Força-bruta.

Ainda na figura 9, percebe-se que o algoritmo de força bruta não é indicado para instâncias grandes como as testadas, pois pode demorar muito a executar. A força bruta só é recomendada em casos onde o número de vértices é pequeno e possui-se limitação de memória, pois as outras variações apesar de mais rápidas, consomem bem mais memória. Em trabalhos posteriores o Força-bruta pode ser melhorado com algumas podas.

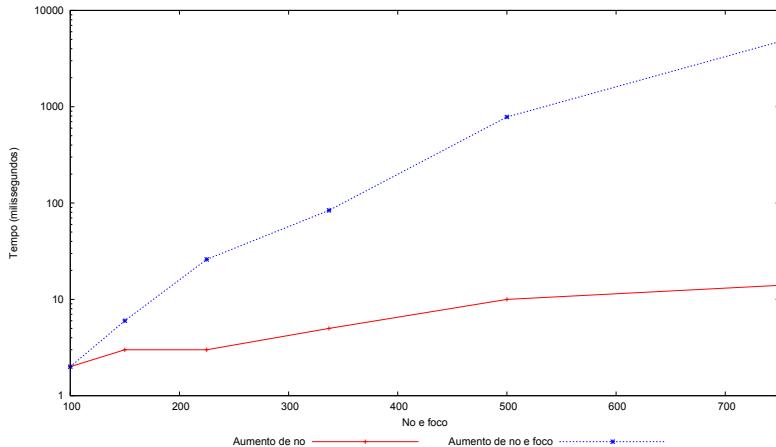


Figura 10. Algoritmo de Programação Dinâmica.

A figura 10 demonstra o comportamento do algoritmo de Programação Dinâmica, onde diferentemente do algoritmo de Força Bruta, mostrado na figura 9, o algoritmo de Programação Dinâmica tem uma dependência maior do numero de focos n_f . Sempre que o número de focos cresce, em realação ao tamanho da entrada as duas curvas se distanciam mais. Infere-se assim que o custo da analise de focos têm um impacto maior neste algoritmo se comparado ao Força-bruta. Assim, quando o número de focos f é garantidamente pequeno em relação ao número de vértices e possui-se memória sobrando, esse algoritmo é mais indicado em relação ao de Força-bruta e o Gulosso.

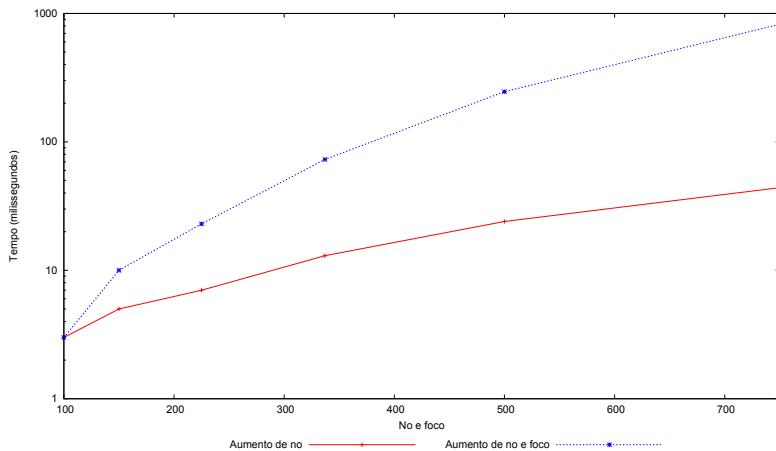


Figura 11. Algoritmo de Gulosso.

Pela figura 11, percebemos que o algoritmo Gulosso tem uma dependência menor do número de focos em relação ao de vértices, se comparado ao algoritmo de de Programação Dinâmica. Percebe-se que as duas linhas se distanciam menos acentuadamente, o que indica que em casos que o número de focos f cresce junto ao de vértices, o algoritmo gulosso parece uma boa opção em relação ao de Programação Dinâmica, principalmente por gastar pouca memória.

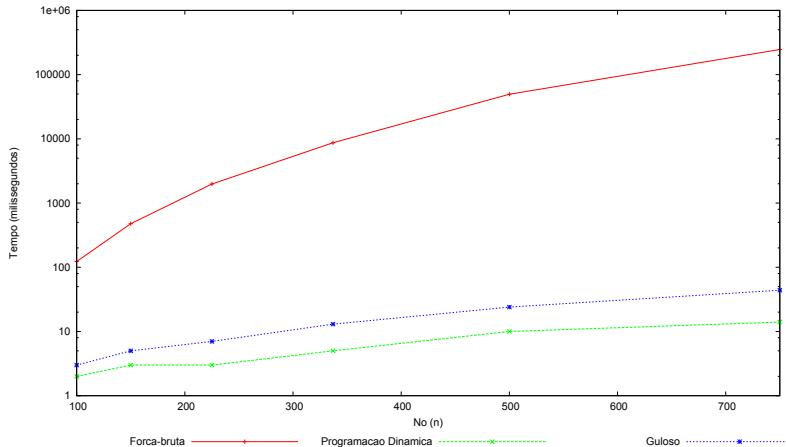


Figura 12. Comparação de tempos dos algoritmos de Foça-bruta, Programação Dinâmica e Guloso em relação ao aumento de vértices, com número de focos fixado em 10

No teste da figura 12, o número de focos foi fixado em 10 e o número de vértices foi aumentado gradativamente ate 750. Percebe-se como o esperado, que o algoritmo de Força-bruta possui um tempo computacional muito alto em relação as duas outras abordagens. Para estes casos quando o número de focos é suficientemente pequeno em relação do de vértices, a abordagem mais indicada é a de programação Dinâmica.

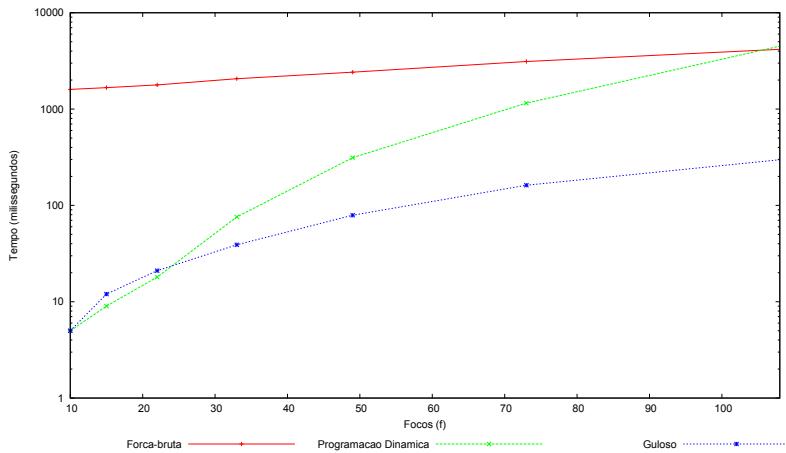


Figura 13. Comparação de tempos dos algoritmos de Foça-bruta, Programação Dinâmica e Guloso em relação ao aumento de focos, onde o número de vértices foi fixado em 200.

No teste da figura 13, o número de vértices foi fixado em 200 e o número de focos diferentes foi aumentado gradativamente até 120. Percebe-se mais uma vez a inviabilidade gerada pelo alto tempo gasto pelo algoritmo de Força-bruta quando o número de focos é pequeno. Para este caso, quando o número de focos cresce rapidamente, o algoritmo de Programação Dinâmica também tem um desempenho ruim, fato ocorre por ele ser muito dependente da verificação dos focos cobertos, como visto na seção 2.2. Para estes casos onde o número de vértices é constante em relação ao de focos, a abordagem mais indicada é a de Programação Dinâmica. Pelos tempos tomados, parece que quando

$n_f > n/2$ o algoritmo de Força-bruta têm um desempenho melhor que o de Programação Dinâmica.

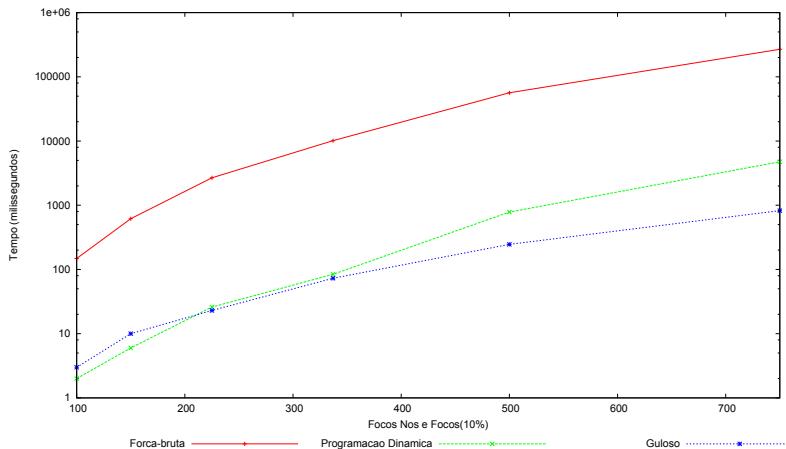


Figura 14. Comparação de tempos dos algoritmos de Foça-bruta, Programação Dinâmica e Guloso em relação ao aumento de focos e nós proporcionalmente.

Os testes mostrados na figura 14, mostram os comportamentos dos algoritmos propostos, quando aumentamos simultaneamente o número de vértices e focos simultaneamente a uma proporção de 10 vértices para 1 foco.

Como pode ser observado na figura 14, para este cenário o algoritmo de Força-bruta possui um tempo computacional muito alto. Neste caso o número de focos é que vai desempatar o melhor algoritmo. Percebe-se que para $n_f < 22$ e $n < 220$ o algoritmo de Programação Dinâmica tem um desempenho melhor, para valores de focos com crescimento suficientemente grande, acima deste ponto o algoritmo mais indicado é o Guloso.

5. Análise de desempenho

Para todos os testes feitos, as saídas de todos os algoritmos foram comparadas entre si, assim acreditamos com base na teoria estudada e adaptada ao problema, todos os três tiveram um resultado ótimo. O algoritmo guloso realmente apresenta ter sub-estrutura ótima, onde dentre os vários testes feitos, nenhum mostrou um contraexemplo para falsear o critério de escolha gulosa adotado.

Para os testes feitos na análise de desempenho, foi usado o cenário onde o número de focos foi aumentado a uma taxa de 10% em relação aos vértices, como pode ser observado no eixo x . O eixo y mostra o tempo de execução dos testes. Para uma normalização dos valores encontrados forma usadas três constantes de aproximação, c_1 , c_2 e c_3 , onde:

- $c_1 = 1/0.00004$
- $c_2 = 1/0.00003$
- $c_3 = 1/0.00001$

Com essa normalização, plotou-se o gráfico do desempenho dos algoritmos, junto a sua complexidade teórica, analisada no exercício 2.

Como o planejado, o algoritmo de Força-bruta, possui complexidade temporal polinomial, e de fato é o mais lento, da ordem de $O(n^3n_f)$. As implementações Gulosa e Dinâmica possuem uma complexidade menor, como o esperado. Onde a implementação Gulosa é da ordem de $O(n^2n_f + m)$ e a implementação Dinâmica é $O(n^2n_f^2)$.

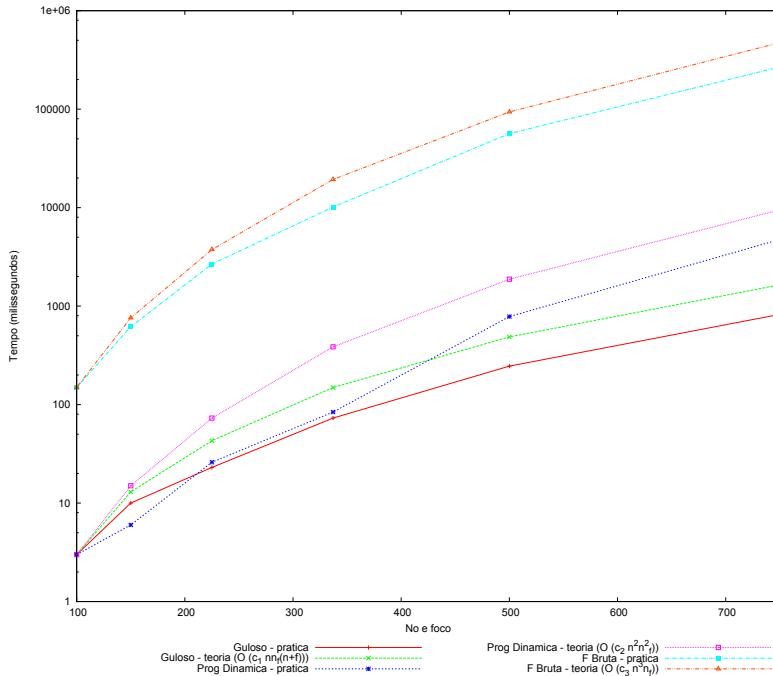


Figura 15. Comparação de tempos dos algoritmos, analisando a complexidade teórica e o comportamento na prática.

Com base na figura 15, podemos perceber que os algoritmos têm um comportamento pouco diferente na prática, se comparado a análise de complexidade temporal feita de forma teórica. Tal fato pode ser ocasionado por vários fatores, dentre eles acreditamos que o que mais influência é que na prática, os algoritmo de Programação Dinâmica e o Guloso acham a solução rapidamente, diminuindo o tempo de execução.

Como podemos verificar visualmente na figura 15, pode-se perceber que todos algoritmos ficaram abaixo de sua complexidade teórica, com um crescimento constante aproximado. Onde acreditamos que essa pode ser usada como um limite assintótico superior.

O algoritmo de Força-bruta foi o que mais ficou próximo de sua complexidade teórica, fato ocorre por ser um algoritmo ingênuo, e sem muitas podas que realmente sejam efetivas para melhora de tempo computacional. Assim o seu desempenho foi bastante similar na prática e na teoria.

O algoritmo Guloso, na prática possui um desempenho melhor que na teoria, com um crescimento assintótico pouco menor que sua complexidade teórica. O algoritmo guloso tende a encontrar a solução bem mais rapidamente que o algoritmo de Força-bruta, como pode ser observado.

Na prática, a partir de um $n > n_0$ o algoritmo Guloso acaba tendo um comporta-

mento melhor que o algoritmo de Programação Dinâmica.

Todos os testes parecem se comportar bem, tendo um crescimento assintótico pouco abaixo que seu limite encontrado na teoria, o que demonstra que a análise feita na seção 15, condiz com o comportamento real de execução do algoritmo.

Referências

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

PAA – Trabalho Prático - Paradigmas

Duílio Campos Sasdelli¹

¹Departamento de Ciência da Computação – UFMG

duilio@dcc.ufmg.br

1. Introdução

O mosquito *Aedes aegypti* é responsável pela transmissão de doenças virais tais como a febre **Zika**, a **Dengue** e a febre **Chikunguya**. A profileração do mosquito pode ser reduzida combatendo-se seus focos de reprodução, os quais, na maioria das vezes, estão localizados em residências ou locais de trabalho.

O objetivo do presente trabalho é selecionar um grupo de voluntários para realização de visitas aos focos do *Aedes aegypti* de tal como que todos os focos sejam visitados pelo menos uma vez. O grupo de voluntários é formado por uma corrente de amigos concebida da seguinte maneira: uma primeira pessoa indica um amigo, que indica outro amigo, e assim sucessivamente, até que o último indique o primeiro.

Mais formalmente, o problema pode ser assim descrito da seguinte forma[Meira 2016]:

Problema ZikaZeroAnelDual

Dados um grafo anel (aquele em que todo vértice conecta-se a exatamente outros dois $\mathbf{G(V,A)}$, em que V é o conjunto de n voluntários e A é o conjunto de $m = |V|$ laços de amizade, um conjunto 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, sendo que $|\mathbf{R(v)}|=2, \forall v \in V$. 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 da solução

Trata-se de um problema de cobertura de conjuntos conectados[Shuai et al. 2006], ou seja, uma variação do problema de cobertura de conjuntos em que se estabelece a seguinte restrição: cada conjunto escolhido deve apresentar algum tipo de relação com pelo menos um dos outros conjuntos escolhidos, isso é, caso a solução não seja unitária. No caso em tela, se um dado voluntário v_i acessa um conjunto de focos F_i e outro voluntário v_j , vizinho de v_i , acessa um conjunto de focos F_j , então F_i e F_j poderiam fazer parte de uma mesma solução, uma vez que ambos há uma relação de amizade entre os dois voluntários.

Foram desenvolvidos três algoritmos com Complexidade Assintótica Polinomial para a resolução do problema, utilizando-se de três paradigmas distintos, quais sejam: **Força Bruta**, **Programação Dinâmica** e **Guloso**. Para as soluções apresentadas, considerou-se que o grafo em anel possui um conjunto de vértices $V = \{v_0, v_1, \dots, v_n\}$ tal que v_0 conecta-se a v_1 , v_1 a v_2 , e assim sucessivamente, até que v_n conecte-se a v_0 e o ciclo se feche.

2.1. Paradigma de Força Bruta

O paradigma de **força bruta** realiza uma busca exaustiva sobre todo o universo de soluções possíveis para o problema escolhendo-se, ao final, aquela que melhor atende às métricas demandadas, ou seja, a solução ótima[Cormen et al. 2009]. Para o problema de cobertura de conjuntos conectados é necessário verificar, circularmente, todas as sequências de conjuntos de tamanho um, depois todas as sequências de tamanho dois, e assim sucessivamente, até que uma solução seja encontrada. Caso uma sequência $\{v_i, v_{i+1}, \dots, v_j\}$ de tamanho $(j - i + 1)$ forme uma solução, não é necessário mais percorrer as sequências de tamanhos maiores, uma vez que essa já é, garantidamente, uma solução de tamanho mínimo.¹

Uma implementação é apresentada no Algoritmo 1:

- **Linhas 2 a 12:** Laço responsável por iterar sobre os diferentes tamanhos de sequência.
- **Linhas 3 a 11:** Laço responsável por iterar sobre os diferentes voluntários que iniciarão uma sequência.
- **Linhas 4 a 7:** Inicializa o conjunto de focos cobertos pela sequência atual (i a $tam - 1$, inclusive) e o preenche com os focos dessa sequência.
- **Linhas 8 a 10:** Se o conjunto de focos cobertos já cobre todo o universo de focos, então uma solução foi encontrada e o procedimento a retorna.

Algorithm 1 Paradigma de Força Bruta

```
1: procedure COBERTURAFORCABRUTA( $V, F, Fc$ )
2:   for  $tam = 1$  to  $|V|$  do
3:     for  $i = 0$  to  $|V| - 1$  do
4:        $FocosCobertos \leftarrow \emptyset$ 
5:       for  $j = 0$  to  $tam - 1$  do
6:          $FocosCobertos \leftarrow FocosCobertos \cup Fc_{(i+j) \% |V|}$ 
7:       end for
8:       if  $|FocosCobertos| == |F|$  then
9:         return  $\{i, (i + tam - 1) \% |V|\}$             $\triangleright$  Retorna índices da solução
10:      end if
11:    end for
12:  end for
13:  return  $null$                                  $\triangleright$  Sem solução
14: end procedure
```

2.1.1. Análise

Em relação ao tempo de execução, o procedimento de união de conjuntos executado na Linha 6 apresenta complexidade assintótica $O(1)$ no pior caso. Isso ocorre porque cada voluntário acessa, no máximo, dois focos e a operação de união de conjuntos, se implementados por meio de uma tabela *Hash*, possuirá complexidade $2 \times O(1)$, uma vez que

¹Se for desejável encontrar a sequência que contém o menor somatório de índices, não será necessário verificar as demais sequências de tamanho $(j - i + 1)$ caso a pesquisa se inicie dos índices mais baixos.

é necessária a inserção de dois elementos. Verifica-se também que o laço que inicia-se na linha 5 é executado um total de tam vezes e os laços que iniciam-se nas linhas 2 e 3 são executados, na pior das hipóteses, $|V|$ vezes cada. Assim, a complexidade assintótica do algoritmo força bruta é dada por:

$$\sum_{tam=1}^{|V|} \sum_{i=0}^{|V|-1} \sum_{j=0}^{tam-1} O(1) = \sum_{tam=1}^{|V|} |V| tam O(1) = \frac{|V|(|V| + 1)}{2} |V| O(1) = O(|V|^3)$$

Em relação à utilização de memória, é necessária a criação de uma estrutura *FocosCobertos*, de tamanho máximo igual a $|F|$ e um vetor de conjunto de focos cobertos por cada voluntário Fc , de tamanho $|F||V|$. Assim, desconsiderando-se a estrutura do grafo, a complexidade de espaço é dada por $O(|F||V| + |F|)$.

2.2. Paradigma de Programação Dinâmica

O problema **ZikaZeroAnelDual** apresenta **subestrutura ótima e sobreposição de subproblemas**, ou seja, é possível de ser desenvolvido um algoritmo de programação dinâmica que o solucione aproveitando-se dessas características. O algoritmo desenvolvido por meio do paradigma de programação dinâmica nada mais é do que uma modificação do paradigma de força bruta que armazena o resultado de operações de união de conjuntos de focos referentes a uma determinada sequência de voluntários.

O paradigma força bruta (Linhas 5 a 7 do Algoritmo 1) realiza (tam) operações de união de conjuntos, a maior parte delas desnecessariamente. Na verdade, só é necessário realizar uma operação de união de conjuntos a cada iteração do segundo laço (Linhas 3 a 11). Isso é possível porque os resultados já foram computados e armazenados em iterações anteriores. A equação de recorrência referente à operação de união de conjuntos de focos cobertos será dada, portanto, por:

$$fc(v_i, v_j) = \begin{cases} fc(v_i, v_{(j-1) \% n}) \cup fc(v_j, v_j) & \text{para } i \neq j \\ Fc_j, & \text{para } i = j \end{cases}$$

A função $fc(v_i, v_j)$ retorna um conjunto de focos cobertos do voluntário de índice i até o voluntário de índice j . Caso i seja igual a j , é retornado o conjunto de focos cobertos por i . Caso i seja diferente de j , é retornada a união do conjunto de focos cobertos de uma sequência de que inicia-se em i e vai até $j - 1$ com os focos cobertos pelo voluntário de índice j .

A equação de recorrência que retorna o menor número de voluntários que cobrem a totalidades de focos é dada por:

$$n_v(V_i, V_j) = \min\{n_v(V_i, V_{(i+k) \% n})\} \text{ para } 0 \leq k \leq j - i, 0 \leq i \leq j \text{ e } |fc(v_i, v_j)| = |F|$$

Assim como no paradigma força bruta, é necessário percorrer todas as sequências de tamanho um, depois todas as sequências de tamanho dois, e assim sucessivamente, até que uma dessas sequências seja capaz de cobrir todos os focos. A diferença é que o resultado das operações de união de conjuntos de uma sequência $\{i, j\}$ será armazenada em uma matriz $|V| \times |V|$ e utilizada para o cálculo de uma sequência $\{i, j + 1\}$. Cada

Algorithm 2 Paradigma de Programação Dinâmica

```
1: procedure COBERTURAPROGRAMACAO(DINAMICA( $V, F, F_c$ )
2:    $F_m[] \leftarrow \emptyset$                                  $\triangleright$  Cria matriz com união dos conjuntos encontrados
3:   for  $i = 0$  to  $|V| - 1$  do
4:      $F_m[i][i] \leftarrow F_{c_i}$ 
5:     if  $|F_{c_i}| == |F|$  then
6:       return  $\{i, i\}$                                  $\triangleright$  Solução com um voluntário: retorna índices
7:     end if
8:   end for
9:   for  $tam = 2$  to  $|V|$  do
10:    for  $i = 0$  to  $|V| - 1$  do
11:       $k \leftarrow (i + tam - 2) \% |V|$                  $\triangleright$  Limite esquerdo
12:       $j \leftarrow (i + tam - 1) \% |V|$                  $\triangleright$  Limite direito
13:       $F_m[i][j] \leftarrow F_m[i][k] \cup F_m[j][j]$ 
14:      if  $|F_m[i][j]| == |F|$  then
15:        return  $\{i, j\}$                                  $\triangleright$  Retorna índices da solução
16:      end if
17:    end for
18:  end for
19:  return  $null$                                  $\triangleright$  Sem solução
20: end procedure
```

elemento $M[i][j]$ da matriz representa a união de conjuntos de uma sequência que se inicia em i e vai até j .

Uma implementação é apresentada no Algoritmo 2:

- **Linhas 2 a 8:** Laço responsável por inicializar a matriz de união de conjuntos encontrados. Primeiramente são armazenados os elementos da diagonal principal, isso é, os conjuntos de focos cobertos por cada voluntário.
- **Linhas 9 a 18:** Laço responsável por iterar sobre os diferentes tamanhos de sequência.
- **Linhas 10 a 17:** Laço responsável por iterar sobre os diferentes voluntários que iniciarão uma sequência.
- **Linhas 11 a 13:** Realiza o cálculo dos focos cobertos pela sequência de voluntários que se inicia em i e vai até $i + tam - 1$. Para tanto, é necessário utilizar o resultado de uniões prévias (sequências $\{i, i+tam-2\}$ e $\{i+tam-1, i+tam-1\}$). Em ambos os casos, o cálculo leva em conta a circularidade do anel e, portanto, é utilizada a divisão modular.
- **Linhas 14 a 16:** Se o conjunto de focos cobertos já cobre todo o universo de focos, então uma solução foi encontrada e o procedimento a retorna.

2.2.1. Análise

O procedimento é muito semelhante ao implementado por meio do paradigma de força bruta. A primeiramente diferença é que, nas Linhas 2 a 8 é realizada a criação de uma

matriz focos cobertos por uma sequência de voluntários. Como a operação possui complexidade de tempo $O(|V|)$, ela não impacta a complexidade do algoritmo. A segunda diferença é que o laço mais interno do procedimento força bruta não é mais necessário, ou seja, só é realizada uma única operação de união de conjuntos de complexidade $O(1)$ no pior caso. Assim, a complexidade assintótica de tempo de execução do algoritmo implementado por programação dinâmica é dada por:

$$\sum_{tam=2}^{|V|} \sum_{i=0}^{|V|-1} O(1) = \sum_{tam=2}^{|V|} |V| O(1) = (|V| - 1)(|V|)O(1) = O(|V|^2)$$

Decorre do resultado acima que a complexidade assintótica de tempo de execução do paradigma de programação dinâmica é $|V|$ vezes mais rápida que o de força bruta. No entanto, como nada na vida vem de graça, o ganho de desempenho traz consigo um custo: maior utilização de memória. Em virtude dos cálculos armazenados, faz-se necessária a criação de uma matriz de conjuntos de focos cobertos de tamanho $|V| \times |V|$, ou seja, a complexidade assintótica de espaço passa a ser $O(|V|^2|F|)$.

2.3. Paradigma Guloso

O paradigma guloso é caracterizado por escolher sempre a opção que parece ser a melhor ou mais vantajosa para um dado momento. Apesar de sempre fazer a escolha ótima, nem sempre o resultado dessas escolhas é a solução ótima, o que é o caso do problema em tela. Para o problema de cobertura de conjuntos conectados, optou-se por utilizar uma heurística que, a partir de uma sequência de voluntários $\{i, j\}$, escolhe o voluntário adjacente $i - 1$ ou $j + 1$ de tal forma que seja maximizado o conjunto de focos cobertos pela nova sequência $\{i - 1, j\}$ ou $\{i, j + 1\}$. Esse procedimento é repetido até que a sequência de voluntários cubra todos os focos. Em ambos os casos, $vEsq$ ou $vDir$ serão atualizados de modo que os limites da sequência sejam acrescidos de um elemento.

Uma implementação do paradigma guloso descrito supra é apresentada no Algoritmo 3:

- **Linhas 2 a 4:** Primeiramente, o conjunto de focos cobertos é inicializado com os focos cobertos pelo voluntário v_0 , o qual comporá a sequência inicial. Os voluntários adjacentes a v_0 serão $v_{|V|-1}$ e v_1 , localizados imediatamente à esquerda ($vEsq$) e à direita ($vDir$) de v_0 , respectivamente.
- **Linhas 5 a 13:** O laço repete-se até que o conjunto de focos cobertos seja igual ao conjunto total de focos.
- **Linhas 6 a 12:** Caso a união do conjunto de focos cobertos com os focos cobertos por v_{vEsq} possua mais elementos que a mesma operação realizada com v_{vDir} , a sequência adicionará os focos cobertos por v_{vEsq} . Caso contrário, serão adicionados os focos cobertos por v_{vDir} .

2.3.1. Análise

O laço que inicia-se na Linha 5 é executado, no pior caso, $|V|$ vezes e, em cada uma delas, realiza duas operações de união de conjuntos de focos. Como cada voluntário acessa, no

Algorithm 3 Paradigma Guloso

```
1: procedure COBERTURAGULOSO( $V, F, Fc$ )
2:    $FocosCobertos \leftarrow Fc_0$ 
3:    $vEsq = (|V| - 1)$ 
4:    $vDir = 1$ 
5:   while  $|FocosCobertos| < |F|$  do
6:     if  $(|(FocosCobertos \cup F_{vEsq})|) > (|(FocosCobertos \cup F_{vDir})|)$  then
7:        $FocosCobertos \leftarrow FocosCobertos \cup F_{vEsq}$ 
8:        $vEsq = (vEsq - 1) \% |V|$ 
9:     else
10:       $FocosCobertos \leftarrow FocosCobertos \cup F_{vDir}$ 
11:       $vDir = (vDir + 1) \% |V|$ 
12:    end if
13:   end while
14:   return  $\{(vEsq + 1) \% |V|, (vDir - 1) \% |V|\}$  ▷ Retorna índices.
15: end procedure
```

máximo, dois focos, cada operação de união de conjuntos, se implementada por meio de uma tabela *Hash*, possuirá complexidade $O(1)$. Assim, tem-se que a complexidade assintótica de tempo de execução do algoritmo guloso é $O(|V|)$.

Em relação à utilização de memória, uma vez que são utilizadas as mesmas estruturas, tem-se que a mesma complexidade assintótica do paradigma força bruta: $O(|F||V| + |F|)$.

3. Testes Realizados

Para a análise empírica, foi utilizado um microcomputador com as seguintes características:

- Processador Intel Core i7, 4790k, 4Ghz, 8Mb de Cache;
- Memória DDR3 8GB;

Foram realizados testes para se averiguar o tempo de execução e qualidade da solução, medida pelo tamanho do conjunto de voluntários encontrados.

3.1. Tempo de Execução

Os testes realizados almejaram avaliar o comportamento dos algoritmos implementados variando-se o número de voluntários, isso é, o tamanho do grafo de anel e o número de focos. Sabe-se que o número de focos $|F|$ deve ser necessariamente maior que 2 e menor que $(2 \times |V|)$. Para todos os testes realizados, foram gerados 1000 grafos aleatórios e tomada a média do tempo de execução, em milissegundos.

O primeiro teste avaliou o comportamento dos algoritmos em relação número de focos $|F|$, métrica que assumiu as seguintes razões do número de voluntários $|V|$: 25%, 50%, 75%, 100%, 125%, 150%, 175%, 200%. Cada teste foi repetido para diferentes valores de $|V|$, quais sejam, 50, 70, 90 e 110. Os resultados podem ser visualizados na Figura 1.

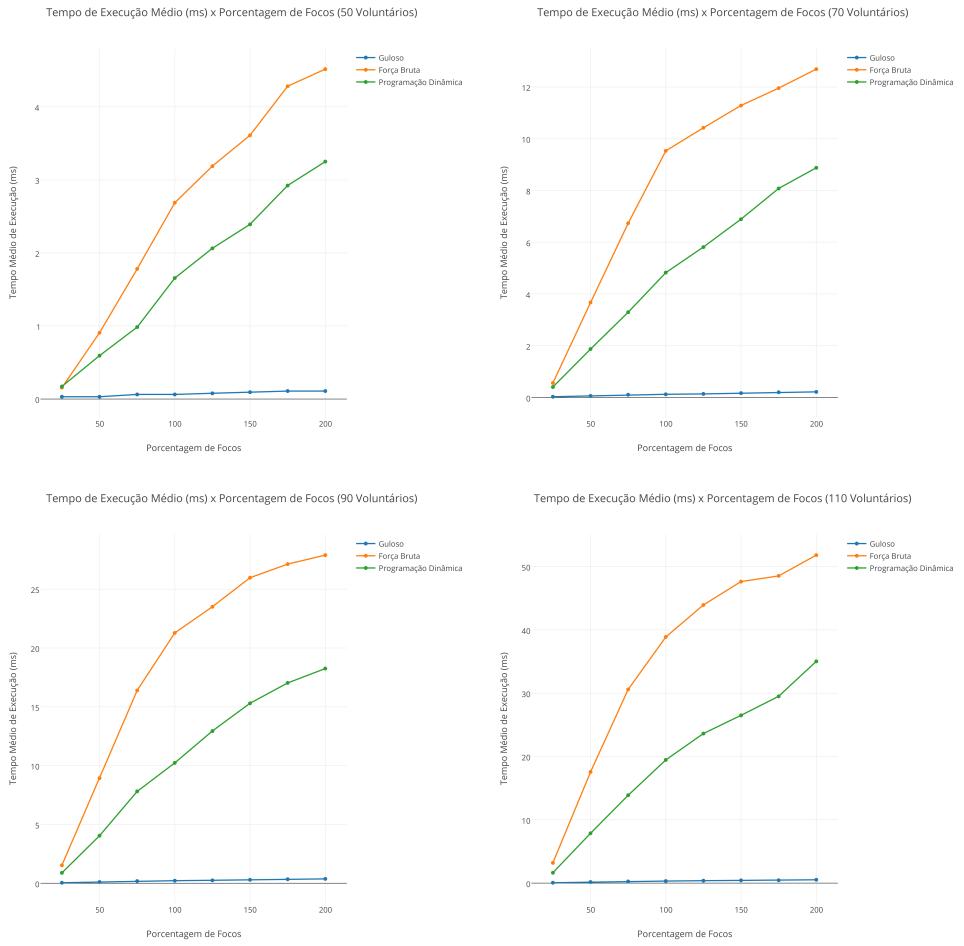


Figura 1. Tempo de execução médio dos algoritmos implementados para diferentes valores de $|F| = |V|/R$ e para $|V| = \{50, 70, 90, 110\}$

Da Figura 1, extrai-se que o tempo de execução para os algoritmos de Força Bruta (FB) e Programação Dinâmica (PD) apresenta um comportamento de acréscimos decrescentes, aparentemente logarítmico, em relação ao número de focos. Isso decorre do fato de que, quanto maior o número de focos, maior a quantidade de sequências de voluntários que devem ser pesquisadas para se encontrar a melhor solução. Se $|F|$ apresentar um valor baixo, a melhor solução será encontrada mais rapidamente, isso é, em uma sequência menor de voluntários. O comportamento do algoritmo guloso (GL) é aparentemente linear em relação ao número de focos, não havendo acréscimos significativos no tempo de execução quando estes aumentam.

O segundo teste avaliou o comportamento dos algoritmos em relação ao número de voluntários $|V|$, métrica que assumiu os seguintes valores: 50, 60, 70, 80, 90, 100, 110, 120. Cada teste foi repetido para os seguintes valores de $|F|$: 50%, 100% e 200%. Os resultados podem ser visualizados na Figura 2.

Conforme esperado, da Figura 2 extrai-se que o tempo de execução apresenta um comportamento aparentemente polinomial para os três algoritmos implementados, isso é, $O(|V|)$ para GL, $O(|V|^2)$ para PD e $O(|V|^3)$ para FB. Vale ressaltar também que, diferentemente de PD e FB, o tempo de execução do algoritmo GL não sofreu grandes

variações para diferentes valores de $|F|$.

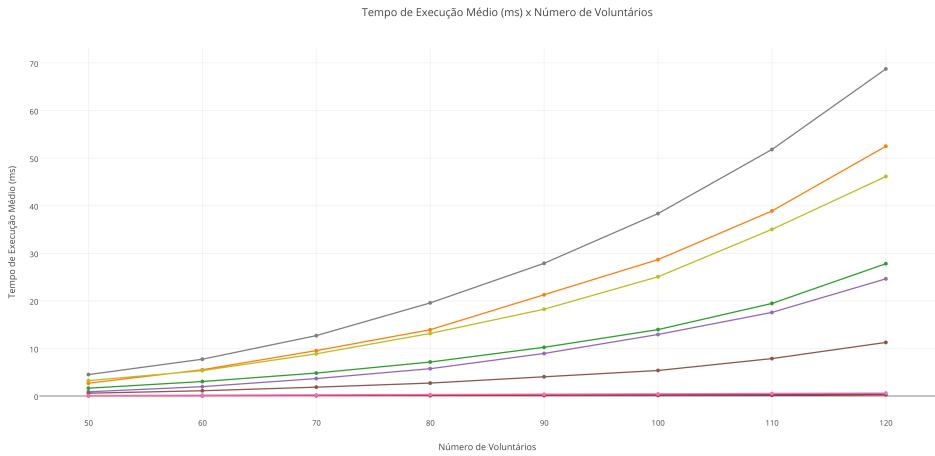


Figura 2. Tempo de execução médio dos algoritmos implementados para diferentes valores de $|V|$

3.2. Qualidade da Solução

Para se atestar a qualidade da solução, pode-se utilizar o tamanho do conjunto de voluntários selecionados. A solução implementada por meio do paradigma guloso, obviamente, por escolher sempre a melhor alternativa em um dado momento, não garante um resultado ótimo. Os paradigmas de FB e PD, por outro lado, garantem sempre a solução ótima. Para o teste de qualidade da solução avaliou-se o comportamento do número de conjuntos à medida que $|F|$ aumenta e aproxima-se de $(2 \times |V|)$. Os resultados podem ser visualizados na Figura 3.

Da Figura 3, extraí-se que o paradigma guloso aproxima-se da solução ótima à medida que $|F|$ aumenta. A explicação para isso é que, à medida que $|F|$ aumenta, é necessário percorrer uma sequência maior de voluntários e, consequentemente, o tamanho do conjunto selecionado. Por exemplo, para $|F| = 2|V|$, deve-se percorrer, obrigatoriamente, toda a sequência de $|V|$ voluntários e, portanto, a solução gulosa é igual à solução ótima. Conclui-se, portanto, que para $|F| > |V|$, caso não seja necessário encontrar a solução ótima, é mais interessante utilizar-se do algoritmo guloso em virtude do menor tempo de execução.

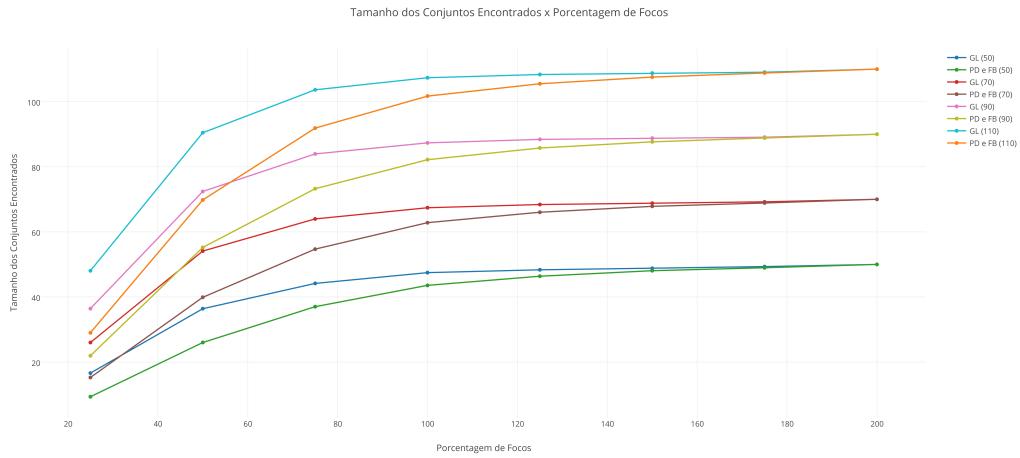


Figura 3. Tamanho dos conjuntos encontrados para diferentes valores de $|F|$

4. Conclusão

No presente trabalho foram implementados algoritmos que solucionam do problema de cobertura de conjuntos conectados, uma variação do problema de cobertura de conjuntos que torna o problema solucionável em tempo de execução polinomial. Não obstante o fato de pertencer à classe de problemas P , foram desenvolvidos não somente uma solução utilizando o paradigma de Força Bruta, mas também outra que faz uso do paradigma de Programação Dinâmica, além de uma terceira com o paradigma Gulosso. Enquanto o paradigma Força Bruta apresentou complexidade assintótica $O(|V|^3)$, o paradigma de Programação Dinâmica apresentou complexidade $O(|V|^2)$ e o paradigma Gulosso, $O(|V|)$.

Apesar de apresentar um tempo de execução menor, o paradigma de Programação Dinâmica necessita maior espaço de memória e, portanto, deve ser descartado se houver restrições em relação à utilização de memória. Já o paradigma Gulosso, apesar de substancialmente mais rápido, não encontra sempre a solução ótima.

Referências

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press.
- Meira, W. (2016). Trabalho prático: Zikazeroaneldual. *Projeto e Análise de Algoritmos (PAA)*.
- Shuai, Tian-Ping, and Hu, X.-D. (2006). Connected set cover problem and its applications. *Algorithmic Aspects in Information and Management*, pages 243–254.

BRUNO LUAN DE SOUSA

RELATÓRIO DO TRABALHO PRÁTICO DE PARADIGMAS

**BELO HORIZONTE
MINAS GERAIS – BRASIL
2016**

Exercício 1 - Proponha algoritmos de Complexidade Assintótica Polinomial com o Tamanho da entrada (n , m e r) para resolver o Problema ZikaZeroAnelDual usando os seguintes paradigmas: Força Bruta, Programação Dinâmica e Algoritmo Gulos.

Antes de mostrar as soluções propostas para os paradigmas exigidos, é importante deixar claro, a estrutura que foi utilizada para a realização deste trabalho. A partir disso, foi mostrado em uma breve explicação, a estrutura utilizada para a implementação dos algoritmos.

Para todos os paradigmas foi usada uma única estrutura que consiste em uma lista de adjacência duplamente encadeada, onde cada voluntário sabe quem é o seu anterior, ou seja, o vértice pelo qual é atingido, e o seu próximo, vértice que ele atinge.

Ainda, foi criado para cada voluntário inferido no problema uma lista para armazenar a solução ótima, utilizada para o paradigma de programação dinâmica. A existência dessa lista terá um melhor entendimento durante a explicação do algoritmo proposto nos paradigmas exigidos.

Com isso, foi proposta solução para a resolução do Problema ZikaZeroAnelDual nos paradigmas: Força Bruta, Gulos e Programação Dinâmica, as quais estão representados e explicados através dos pseudocódigos a seguir.

Força Bruta

A solução de Força Bruta proposta para resolução do problema baseou-se em uma combinação de voluntários conexos de forma que essa combinação fosse controlada por um nível, denominada no algoritmo por um atributo k . Assim, para um determinado nível, são testadas combinações de forma que não haja repetição. Para evitar esse problema, dado um nível k , o algoritmo gera combinações de tamanho k , do elemento voluntário inicial até o elemento $k-1$ a sua frente. Essas combinações são rotacionadas pelo grafo de forma que o voluntário inicial para a próxima combinação seja o elemento posterior ao voluntário inicial atual. Esse processo ocorre até que todos os voluntários do grafo sejam elemento inicial de uma combinação neste nível. Quando isso acontece e o algoritmo não encontra a solução ótima, a formação de combinações para esse nível é finalizado e recomeça para um nível superior.

A **Figura 1** mostra o pseudocódigo que ilustra a solução proposta para esse paradigma.

Algorithm 1 Algoritmo de Força Bruta para resolver o problema do Zica

```

1: função ALGORITMOFORCABRUTA
2:   solucao ← ∅
3:   para k ← 1 até (voluntarios.length) faça
4:     i ← 0
5:     voluntarioInicial ← v0
6:     v ← voluntarioInicial
7:     enquanto i < voluntarios.length faça
8:       solucao ← solucao ∪ {k elementos com inicio em voluntarioInicial}
9:       se verificaSolucaoOptima(solucao) então
10:         mostraSolucaoOptima(solucao)
11:         finaliza função
12:       senão
13:         se k = voluntarios.length então
14:           solucao ← ∅
15:           finaliza função
16:         senão
17:           i ++
18:           solucao ← ∅
19:           voluntarioInicial ← voluntarioInicial.proximo
20:         fim se
21:       fim se
22:     fim enquanto
23:   fim para
24: fim função

```

Figura 1. Pseudocódigo do algoritmo de força bruta.

O algoritmo inicia definindo uma lista de solução vazia, linha 2. O laço de repetição for (para), linha 3-23, é responsável por fazer o controle do nível em que o algoritmo vai pesquisar a solução. Assim, se k é 1, então o algoritmo gera soluções de tamanho 1, se k é 2, o algoritmo gera soluções de tamanho 2, e assim por diante.

O laço de repetição enquanto, (linha 7-17) é responsável por garantir que a formação de combinações para um determinado nível, tem como elemento inicial cada voluntário do grafo. Isso ocorre para garantir que sejam geradas todas as combinações para cada nível sem que haja repetições de combinações. Na linha 8, o algoritmo gera uma solução de tamanho k, correspondente ao nível atual, na qual está contida o elemento *voluntarioInicial* e os k-1 elementos a sua frente. Na linha 9, é verificada se os voluntários contidos na solução atende a todos os focos. Caso atenda, é mostrada os elementos da solução e o algoritmo é finalizada, caso contrário, o algoritmo pode tomar duas decisões. Se o valor de k não for igual à quantidade de voluntário do grafo, linha (16), o algoritmo prepara o ambiente para gerar uma nova combinação para este nível atual, a qual começará do elemento posterior a elemento *voluntarioInicial* atual. Por outro lado, se o valor de k igual à quantidade de voluntário do grafo, o algoritmo chegou no último nível de combinações possível. Nesse nível é gerada uma combinação com todos os voluntários desse grafo, e qualquer outra combinação que for gerada após essa, haverá repetições. Para evitar isso, quando essa condição é atendida, e não é encontrada uma solução, o algoritmo finaliza sua execução, uma vez que todas as combinações possíveis já foram testadas, e retorna vazio para o usuário, pois para essa instância do problema não existe uma solução.

Algoritmo Guloso

A solução gulosa proposta para a resolução deste problema, baseou-se na escolha dos elementos vizinhos que aumentam o número de focos atendidos. A partir disso, o algoritmo começa com o elemento inicial verificando tanto o seu anterior quanto o seu posterior. Aquele que aumentar o número de focos atendidos, o algoritmo adiciona esse

elemento na lista de solução. Para os próximos elementos, o algoritmo analisa apenas o próximo ou o anterior, e estes adicionados no final da lista de solução ou no início da lista de solução, respectivamente. Em caso de empate, ou seja, se os dois elementos analisados contribuírem com o mesmo valor para o aumento dos focos atendidos, o algoritmo seleciona os dois, caso eles não sejam o mesmo elemento, ou seleciona apenas um, caso sejam o mesmo elemento. No final é realizado um refinamento dessa solução para retirar elementos redundantes. Porém, para a implementação desse paradigma, abriu-se mão da otimalidade da solução para que o algoritmo possa ter um tempo de execução menor. Assim, a solução gerada no final da execução do algoritmo não é garantida de sempre ser a solução ótima para o problema.

A **Figura 2** mostra o pseudocódigo que ilustra a solução proposta para esse paradigma.

Algorithm 2 Algoritmo Guloso para resolver o problema do Zica

```

1: função ALGORITMOGULOSO
2:    $v_{Incial} \leftarrow v_0$ 
3:    $solucao \leftarrow v_0$ 
4:    $vEsquerda \leftarrow v_{Incial.anterior}$             $\triangleright vEsquerda e o anterior de v$ 
5:    $vDireita \leftarrow v_{Incial.proximo}$             $\triangleright vDireita e o proximo a v$ 
6:   enquanto  $vEsquerda \notin solucao$  e  $vDireita \notin solucao$  faz
7:     se verificaSolucaoOptima(solucao) então
8:       finaliza o laço enquanto
9:     senão
10:       $esquerda \leftarrow acrescentaSolucao(vEsquerda, solucao)$ 
11:       $direita \leftarrow acrescentaSolucao(vDireita, solucao)$ 
12:      se esquerda = direita e  $vEsquerda \neq vDireita$  então
13:        adciona  $vEsquerda$  no inicio de solucao
14:        adciona  $vDireita$  no final de solucao
15:         $vEsquerda \leftarrow vEsquerda.anterior$ 
16:         $vDireita \leftarrow vDireita.proximo$ 
17:      senão
18:        se  $vEsquerda \geq vDireita$  então
19:          adciona  $vEsquerda$  no inicio de solucao
20:           $vEsquerda \leftarrow vEsquerda.anterior$ 
21:        senão
22:          adciona  $vDireita$  no final de solucao
23:           $vDireita \leftarrow vDireita.proximo$ 
24:        fim se
25:      fim se
26:    fim se
27:  fim enquanto
28:   $solucao \leftarrow refinaSolucao(solucao)$ 
29:  mostraSolucaoOptima(solucao)
30: fim função

```

Figura 2. Pseudocódigo do algoritmo guloso.

O algoritmo inicia definindo alguns objetos importantes para o decorrer da execução. É definido o voluntário v_0 como elemento inicial e o anterior e posterior a esse elemento inicial como voluntário Esquerda e voluntário Direita. Esses dois últimos objetos terão uma função importante no algoritmo, pois a escolha de um dos dois garante que a solução não fique desconexa.

O laço de repetição while (enquanto), linha 7-28, garante a verificação de todos os voluntários na criação da solução para um v_i . Dentro desse laço, é verificado se a solução do elemento atual atinge todos os focos. Caso atenda, o laço while (enquanto) é finalizado. Caso não atinja, o algoritmo analisa a quantidade de focos que o voluntário Esquerda atende e a quantidade de focos que o voluntário Direita atende. Aquele que maximiza a quantidade de focos, é adicionado no início, se o voluntário Esquerda atenda mais focos,

ou no final da solução, se o voluntário Direita anteda mais focos, e o valor do escolhido é alterado para o elemento da esquerda ou elemento da direita, de acordo com a escolha, linha 20-21 e linha 23-24. Em caso de empate e os elementos não serem iguais, ambos são adicionados na solução e mudam seus valores, linha 14-17. Em caso de empate e os elementos serem iguais, apenas um deles é adicionado na solução, linha 20-21.

Ao final do laço de repetição while (enquanto), linha 29, é realizado um refinamento da solução para garantir que as redundâncias sejam eliminadas.

Por fim, ao finalizar o refinamento da solução, o algoritmo exibe para o usuário a solução final encontrada, que não é garantida de ser a solução ótima para o problema.

Programação Dinâmica

A solução de programação dinâmica proposta para a resolução deste problema, baseou-se em uma estratégia de inserção dos elementos vizinhos, tanto da direita quanto da esquerda na possível solução para o problema. Cada vez que há uma inserção de elementos na solução, o algoritmo verifica se todos os focos são atendidos. Caso não atenda, o processo é repetido até encontrar um conjunto de voluntários conexo que atenda todos os focos. Ao encontrar uma solução para um determinado voluntário, essa solução é refinada para que sejam retirados elementos redundantes. Esse processo se repete para todos os elementos do grafo, e ao final da solução é verificada a solução de menor tamanho, a qual é retornada pelo algoritmo como solução ótima.

A **Figura 3** mostra o pseudocódigo que ilustra a solução proposta para esse paradigma.

Algorithm 3 Algoritmo de Programação Dinâmica para resolver o problema do Zica

```

1: função ALGORITMOPROGRAMACAO DINAMICA
2:   para i ← 0 até (voluntarios.length – 1) faça
3:     vIncial ← vi
4:     vIncial.S ← vi           ▷ S é a solução Ótima para cada voluntário
5:     vEsquerda ← vIncial.anterior      ▷ vEsquerda é o anterior de v
6:     vDireita ← vIncial.proximo        ▷ vDireita é o próximo a v
7:     enquanto vEsquerda ≠ vIncial.S e vDireita ≠ v.Inicial faça
8:       se verifica(vIncial.S) então
9:         finaliza o laço enquanto
10:        senão
11:          se vEsquerda = vDireita.S então
12:            adiciona vDireita no final de vIncial.S
13:            vDireita ← vDireita.proximo
14:          senão
15:            adiciona vEsquerda no inicio de vIncial.S
16:            adiciona vDireita no final de vIncial.S
17:            vEsquerda ← vEsquerda.anterior
18:            vDireita ← vDireita.proximo
19:          fim se
20:        fim se
21:      fim enquanto
22:      se verifica(vIncial.S) então
23:        vIncial.S ← refinaSolucao(v.Inicial)
24:      senão
25:        v.Inicial.S ← ∅
26:      fim se
27:    fim para
28:    solucaoFinal ← solucaoMenorTamanho(voluntarios)
29:    mostraSolucaoÓtima(solucaoFinal)
30:  fim função

```

Figura 3. Pseudocódigo do algoritmo de programação dinâmica.

O algoritmo inicia com um laço for (para), linha 2-27, responsável por garantir e que será buscada uma solução para cada voluntário que compõe o grafo. A partir disso, as linhas 3-6 definem o elemento v_i como elemento inicial e o anterior e posterior a esse elemento inicial como voluntário Esquerda e voluntário Direita. Esses dois últimos objetos garantem que a solução não fique desconexa.

O laço de repetição while (enquanto), linha 7-21, garante que os elementos, ainda não descobertos, possam ser adicionados à lista S do voluntário v_i até formar uma solução para o problema. Dentro desse laço, é verificado se a solução do elemento atual atinge todos os focos. Caso atenda, o laço while (enquanto) é finalizado. Caso não atenda, tanto o elemento vEsquerda quanto o elemento vDireita são adicionados na solução, se não forem iguais. Em caso de igualdade é adicionado apenas um dos dois para evitar repetição.

Ao final do laço while (enquanto), a solução é verificada novamente para garantir que esta é válida. Caso seja válida, ela é refinada para que elementos redundantes sejam retirados. Caso não seja válida, ela é limpada, e o processo inicia-se para outro elemento do grafo.

Por fim, ao finalizar o laço de repetição for (para), linha 28-29, é realizada pesquisa nos elementos do grafo a procura da solução de menor tamanho, a qual é definida como solução final, e esta é mostrada para o usuário como a solução ótima para este algoritmo.

Exercício 2 - Analise as complexidades Temporais e Espaciais (usando Notação Assintótica) dos algoritmos propostos no Exercício 1.

Força Bruta

Análise Temporal

Realizando a análise das funções mostradas no pseudocódigo no exercício anterior, tem-se o seguinte:

- Na linha 2, o algoritmo possui um laço de repetição responsável por definir o tamanho das combinações que serão feitas durante a execução do algoritmo. Esse laço começa em um e aumenta gradativamente, de acordo com que as soluções não são encontradas. No pior caso, esse laço pode atingir o número total de voluntários existentes no grafo. Portanto, o tempo de execução desse laço inicial do algoritmo é: $T(n) = v$, onde v é a quantidade total de voluntários existentes no grafo.
- Na linha 9, existe uma função chamada verificaSolucaoOtima. Essa função percorre uma lista passada como parâmetro, e considerada como a possível solução, comparando a conexão de cada voluntário dessa lista com cada foco existente no grafo para saber se essa lista consegue cobrir todos os focos. O tempo gasto para realizar essa verificação no pior caso é $v*f$, onde v é o número de voluntário e f é o número de focos. Porém, como essa função está dentro de um outro laço de repetição, enquanto, que no pior caso também atinge o número total de vértice (v), a complexidade total da execução dessa função é: $T(n) = v*(v*f) = v^2*f$.
- Na linha 8 é formada a combinação das soluções de acordo com o nível estabelecido pela variável k . Para cada valor de k , o algoritmo gera uma combinação iniciando de um dos elementos do grafo. A variável i , presente no laço de repetição while, é responsável por garantir que tenham exatamente v combinações por nível. Porém, no último nível ($i = v$), é realizada uma única combinação. Assim, a complexidade total dessa linha é: $T(n) = v^2 - v + 1$.
- As complexidades da linha 9 e da linha 8 estão sendo executadas uma após a outra,

então elas devem ser somadas. Assim, temos: $T(n) = (v^2*f) + (v^2-v+1) = v^2*f + v^2 - v + 1$.

- Por fim, como a complexidade do item anterior está sendo executada dentro de um laço de repetição para com complexidade $T(n) = v$, ela deve ser multiplicada pela complexidade do laço, resultando em: $T(n) = v*(v^2*f + v^2 - v + 1) = v^3*f + v^3 - v^2 + v$. Logo, a ordem de complexidade temporal desse algoritmo é $O(v^3)$.

Análise Espacial

Realizando a análise das funções mostradas no pseudocódigo no exercício anterior, tem-se o seguinte:

- Na montagem da solução, é criada uma lista, onde são adicionados os elementos candidatos a solução e verificados. Caso essa solução criada não atenda todos os focos, a lista é esvaziada para uma nova criação. Baseado nisso, o maior tamanho que essa lista pode assumir, é quando todos os voluntários do grafo forem armazenados nela como uma possível solução, ou seja, v . Assim, a complexidade espacial para essa lista de solução é: $S(n) = v$.
- Outro ponto do algoritmo em que há um grande uso de memória é quando a solução é comparada, a função verifica. Nessa função, a lista de possível solução é passada para a como parâmetro e verificada se atende a todos os focos. Como falado no item anterior a lista de possível solução tem no seu pior caso tamanho v . E a lista de focos do algoritmo possui tamanho f . Logo, a complexidade espacial dessa função é $S(n) = v*f$.
- Por fim, somando as duas funções de complexidade, temos: $S(n) = v + v*f$. Logo, a ordem de complexidade espacial desse algoritmo é $O(v)$.

Algoritmo Guloso

Análise Temporal

Realizando a análise das funções mostradas no pseudocódigo no exercício anterior, tem-se o seguinte:

- Na linha 8, existe uma função chamada verificaSolucaoOtima. Essa função percorre uma lista passada como parâmetro, e considerada como a possível solução, comparando a conexão de cada voluntário dessa lista com cada foco existente no grafo para saber se essa lista consegue cobrir todos os focos. O tempo gasto para realizar essa verificação no pior caso é $v*f$, onde v é o número de voluntário e f é o número de focos. Porém, como essa função está dentro de um outro laço de repetição, enquanto, que no pior caso também atinge o número total de vértice (v), a complexidade total da execução dessa função é: $T(n) = v*(v*f) = v^2*f$.
- Na linha 29, é realizado um refinamento para poder eliminar as estruturas redundantes da solução final para cada vértice. Essa função tem um tempo de complexidade $T(n) = 2*v$.
- Como a linha 8 e a linha 29 não estão dentro de nenhum laço de repetição, essas duas funções de complexidades podem ser somadas, gerando a função de complexidade: $T(n) = v^2*f + 2*v$. Logo, como não há nenhuma outra linha desse algoritmo que exija um alto custo de tempo de execução, a ordem de complexidade temporal desse algoritmo é $O(v^2)$.

Análise Espacial

Realizando a análise das funções mostradas no pseudocódigo no exercício anterior, tem-se o seguinte:

- Na montagem da solução, é criada uma lista, onde são adicionados os elementos candidatos a solução e verificados. Caso essa solução criada não atenda todos os focos, a lista é esvaziada para uma nova criação. Baseado nisso, o maior tamanho que essa lista pode assumir, é quando todos os voluntários do grafo forem armazenados nela como uma possível solução, ou seja, v . Assim, a complexidade espacial para essa lista de solução é: $S(n) = v$.
- Outro ponto do algoritmo em que há um grande uso de memória é quando a solução é comparada, a função verifica. Nessa função, a lista de possível solução é passada para a como parâmetro e verificada se atende a todos os focos. Como falado no item anterior a lista de possível solução tem no seu pior caso tamanho v . E a lista de focos do algoritmo possui tamanho f . Logo, a complexidade espacial dessa função é $S(n) = v*f$.
- Por fim, somando a função de complexidade do item anterior com a função de complexidade do primeiro item, temos: $S(n) = v + v*f$. Logo, a ordem de complexidade espacial desse algoritmo é $O(v*f)$.

Programação Dinâmica

Análise Temporal

Realizando a análise das funções mostradas no pseudocódigo no exercício anterior, tem-se o seguinte:

- Na linha 2, o algoritmo possui um laço de repetição responsável por garantir que seja achada uma solução ótima para voluntário do grafo. Esse laço começa em zero e finaliza em $v - 1$. Portanto, o tempo de execução desse laço inicial do algoritmo é: $T(n) = v$, onde v é a quantidade total de voluntários existentes no grafo.
- Na linha 8, existe uma função chamada verifica. Essa função percorre uma lista passada como parâmetro, e considerada como a possível solução, comparando a conexão de cada voluntário dessa lista com cada foco existente no grafo para saber se essa lista consegue cobrir todos os focos. O tempo gasto para realizar essa verificação no pior caso é $v*f$, onde v é o número de voluntário e f é o número de focos. Porém, como essa função está dentro de um outro laço de repetição, enquanto, que no pior caso também atinge o número total de vértice (v), a complexidade total da execução dessa função é: $T(n) = v*(v*f) = v^2*f$.
- Na linha 22 existe novamente uma comparação usando a função verifica. Logo a complexidade temporal nessa linha é: $T(n) = v*f$.
- Na linha 23, é realizado um refinamento para poder eliminar as estruturas redundantes da solução final para cada vértice. Essa função tem um tempo de complexidade $T(n) = 2*v$.
- Como a linha 8, a linha 22 e a linha 23 estão dentro do laço de repetição para, linha 2, essas duas funções de complexidades podem ser somadas, gerando a função de complexidade: $T(n) = v^2*f + 2*v + v*f$
- Como falado no item anterior, as linhas 8, 22 e 29 estão dentro de um laço de repetição, linha 2, que tem como complexidade tempo v . Com isso a função de

- complexidade derivada no item anterior é executada v vezes. Logo, a nova função de complexidade é: $T(n) = v*(v^2*f + 2*v + v*f) = v^3*f + 2*v^2 + v^2*f$.
- Por fim, no final do laço de repetição para, linha 28, existe uma função que busca entre todos os voluntários existentes no grafo, a solução de menor tamanho. Essa função tem um tempo de complexidade v . Assim, somando essa complexidade com a resultado do laço de repetição, linha 2-27, temos: $T(n) = (v^3*f + 2*v^2 + v^2*f) + v = v^3*f + 2*v^2 + v^2*f + v$. Logo, a ordem de complexidade temporal desse algoritmo é $O(v^3)$.

Análise Espacial

Realizando a análise das funções mostradas no pseudocódigo no exercício anterior, tem-se o seguinte:

- Na montagem da solução, é criada uma lista, onde são adicionados os elementos candidatos a solução e verificados. Caso essa solução criada não atenda todos os focos, a lista é esvaziada para uma nova criação. Baseado nisso, o maior tamanho que essa lista pode assumir, é quando todos os voluntários do grafo forem armazenados nela como uma possível solução, ou seja, v . Assim, a complexidade espacial para essa lista de solução é: $S(n) = v$.
- Outro ponto do algoritmo em que há um grande uso de memória é quando a solução é comparada, a função verifica. Nessa função, a lista de possível solução é passada para a como parâmetro e verificada se atende a todos os focos. Como falado no item anterior a lista de possível solução tem no seu pior caso tamanho v . E a lista de focos do algoritmo possui tamanho f . Logo, a complexidade espacial dessa função é $S(n) = v*f$. Como existem dois pontos no código que essa função existe, a complexidade total dessa função é: $S(n) = 2*v*f$.
- Como foi falado no primeiro item, é criada uma lista de solução que no pior caso tem tamanho v . Porém, essa lista é criada para cada vértice e armazenada respectivamente nos mesmos. Assim, temos como função de complexidade: $S(n) = v*v = v^2$.
- Por fim, somando a função de complexidade do item anterior com a função de complexidade do segundo item, temos: $S(n) = v^2 + 2*v*f$. Logo, a ordem de complexidade espacial desse algoritmo é $O(v^2)$.

Exercício 3 - Implemente os algoritmos propostos no Exercício 1 na linguagem de programação C, C++, Java ou Python.

A implementação dos algoritmos propostos no Exercício 1 foi realizado na linguagem Java contendo 5 classes: Foco, Grafo, Leitura, TrabalhoParadigmas (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 4 - Execute testes da implementação do Exercício 3, propondo instâncias interessantes para o Problema ZikaZeroAnelDual. Uma sugestão é que seja produzido um gráfico do Tempo de execução por Tamanho da entrada (n, m e r) comparando os três algoritmos implementados.

Para realização dos testes, foram utilizadas dez instâncias de diferentes tipos, e para

cada instância execução, foi medido o tempo de execução do algoritmo através de uma variável de tempo. Essa variável inicia a contagem do tempo antes do algoritmo começar sua execução. Quando o algoritmo finaliza sua execução, essa variável é finalizada também, e é verificado o tempo gasto pelo algoritmo em milissegundos.

Os testes realizados nesse exercício foram executados em uma máquina Dell, modelo Dell Inspiron, com a seguinte configuração: 8 GB de memória RAM, 1 TB de HD e processador i7. Foram utilizadas instâncias de diferentes tamanhos para poder analisar em quais situações, um algoritmo consegue ser mais rápido que o outro. A **Figura 4** mostra um gráfico com os resultados do tempo de execução dos algoritmos, representado em milissegundos (ms).

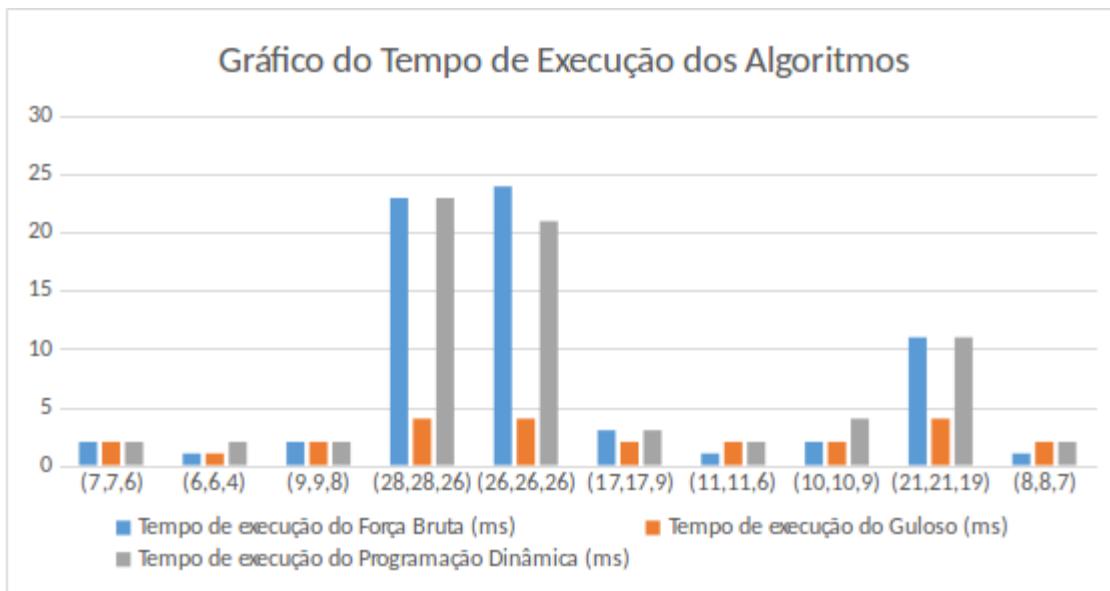


Figura 4. Gráfico de tempo de execução dos algoritmos.

O gráfico da **Figura 4**, 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 milissegundos (ms). A entrada está representada pelo eixo x e o tempo de execução está representado pelo eixo y. Analisando o gráfico, de início é possível observar que na maioria dos casos executados, o algoritmo guloso foi mais rápido que o algoritmo de força bruta e o algoritmo de programação dinâmica. Isso ocorreu devido ao fato da complexidade do algoritmo guloso ser menor que a complexidade dos demais. Porém, como já foi falado, para fazer um algoritmo com um tempo de execução mais rápido, abriu-se mão da otimalidade da solução. Assim, o algoritmo guloso, encontra uma solução para o problema, porém, sem garantias que essa solução seja a melhor para a instância fornecida de entrada.

Em alguns casos, como na execução da sexta instância (11,11,6) e da oitava instância (8,8,7), o algoritmo de força bruta teve um tempo de execução menor que o algoritmo guloso. Isso ocorreu devido a dois fatores. Primeiro, em instâncias pequenas, ambos não possuem diferença em seu tempo de execução e o segundo fator é que dependendo de onde a solução para o problema está situada, o algoritmo de força bruta consegue encontrá-la mais rápido que o algoritmo guloso. Por isso é interessante que haja uma variabilidade nas amostras utilizadas nos testes, a fim de analisar como será o comportamento dos algoritmos quando executados para instâncias grandes.

Outra comparação analisada foi com relação a execução do algoritmo de força bruta e o algoritmo de programação dinâmica. Como para esses dois algoritmos não se abriu mão da otimalidade da solução, eles tiveram uma ordem de complexidade maior que o algoritmo guloso. Analisando o gráfico, percebe-se que houve uma igualdade no tempo final de ambos algoritmos, o que já era esperado devido a ordem de complexidade dos mesmos. Para algumas instâncias específicas no teste um algoritmo foi melhor que o outro, porém, o que influenciou nesses resultados foi a localização da solução na instância para o qual o teste foi executado. Em instâncias, cuja solução possui poucos elementos, o algoritmo de força bruta executa mais rápido, em instâncias, cuja solução possui muitos elementos, o algoritmo de programação dinâmica executa mais rápido.

Exercício 5 - Compare a análise e a execução, respectivamente, Exercícios 2 e 4. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

Após a execução dos testes no exercício 4, percebe-se que o algoritmo guloso teve uma execução em um curto espaço de tempo, porém, a solução gerada não tem garantias de otimalidade. Os demais algoritmos, força bruta e programação dinâmica, garantem que a solução encontrada é a solução ótima, porém para instâncias muito grande, esses algoritmos gastam um grande tempo de execução, no qual um executará mais rápido que o outro dependendo de como a solução ótima está posicionada no grafo.

Ao realizar uma análise de complexidade, exercício 2, percebe-se que o algoritmo de força bruta, para encontrar a solução ótima para uma respectiva instância fornecida para o problema, possui uma ordem de complexidade $O(v^3)$. O algoritmo guloso, possui uma ordem de complexidade menor, $O(v^2)$, porém não garante que a solução encontrada no final de sua execução, seja a solução ótima. Por fim, o algoritmo de programação dinâmica, também possui ordem de complexidade $O(v^3)$, e consegue encontrar a solução ótima para o problema.

Por fim, ao comparar a ordem de complexidade dos algoritmos e os resultados reais obtidos para as instâncias testadas neste trabalho, percebeu-se que a diferença de complexidade influencia muito na execução do algoritmo. Na maioria dos casos de testes executados, a ordem de complexidade quadrática foi melhor que a ordem de complexidade cúbica. Nos casos em que o cúbico executou mais rápido, o principal motivo para esse ocorrido se deve ao fato do tamanho das instâncias utilizadas nos testes, as quais possuíam pouca quantidade de focos e voluntários. Quando executado os testes para instâncias maiores, ou seja, como maior número de voluntários compondo o grafo, maior número de focos a serem cobertos e uma maior distribuição dos focos entre os voluntários, pode-se confirmar que a execução quadrática foi muito mais rápida que a execução cúbica, validar a concordância entre o tempo de execução dos algoritmos e as funções de complexidades extraídas no exercício 2.

Paradigmas de Projeto de Algoritmos: Zika Zero Anel Dual

Data: 17 de junho de 2016

Roberta Coeli Neves Moreira

Exercício 1

Proponha algoritmos de Complexidade Assintótica Polinomial com o tamanho da entrada (n , m e r) para resolver o Problema ZikaZeroAnelDual, usando os seguintes paradigmas:

- **Busca por Força Bruta:** o pseudo-código para o paradigma de busca por força bruta para resolver o problema é apresentado no Algoritmo 1. O algoritmo basicamente percorre o grafo G a partir de cada vértice $v \in \mathbb{V}$, verificando o menor grupo de vértices que contém todos os focos a partir daquele vértice. Se o grupo encontrado for menor do que menor global até o momento (*best_vertices*), *best_vertices* recebe seu valor. Caso contrário, a busca continua até que todas as possibilidades tenham sido verificadas.

É importante salientar que a busca em profundidade (DEPTH-FIRST-SEARCH) (linha 5) é feita apenas com a finalidade de obter uma direção única para verificar os vértices do grafo, o qual é cíclico.

Algoritmo 1: ZIKAZEROANELDUAL - Busca por Força Bruta

Data: $G = (\mathbb{V}, \mathbb{A}, \mathbb{F}, \mathcal{R})$
Result: *best_vertices*, where *best_vertices* 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     best_vertices  $\leftarrow$  list containing all vertices
5     ordered_vertices  $\leftarrow$  DEPTH-FIRST-SEARCH( $\mathbb{V}, \mathbb{A}$ )
6     foreach  $v \in \text{ordered\_vertices}$  do
7          $\mathbb{V}_{\text{current}} = \{v\}$ 
8          $\mathbb{F}_{\text{current}} = \{\mathcal{R}(v)\}$ 
9         while  $\mathbb{F}_{\text{current}}$  does not contain all focuses and not all the vertices were checked do
10             $next_v \leftarrow$  next element in ordered_vertices
11            add  $next_v$  to  $\mathbb{V}_{\text{current}}$ 
12            add  $\mathcal{R}(next_v)$  to  $\mathbb{F}_{\text{current}}$ 
13        end
14        if  $|\mathbb{V}_{\text{current}}| < |\text{best\_vertices}|$  then
15             $\text{best\_vertices} \leftarrow \mathbb{V}_{\text{current}}$ 
16        else if  $|\mathbb{V}_{\text{current}}| == |\text{best\_vertices}|$  and sum of indexes in  $\mathbb{V}_{\text{current}}$  is smaller than sum of
17            indexes in best_vertices then
18                 $\text{best\_vertices} \leftarrow \mathbb{V}_{\text{current}}$ 
19            end
20        end
21    return best_vertices
21 end
```

- **Programação Dinâmica:** o pseudo-código para o paradigma de Programação Dinâmica para resolver o problema é apresentado no Algoritmo 2. O algoritmo basicamente verifica os focos acessados por um conjunto de vértices, o qual aumenta de tamanho a cada iteração. Os valores calculados são armazenados em uma matriz *opt* de tamanho $n \times n \times r$.

Para o problema, o algoritmo de Programação Dinâmica (PD) foi implementado de forma iterativa e *bottom-up*. Nesse algoritmo, a subestrutura ótima consiste em computar os focos que podem ser alcançados a partir de um vértice v na ordem estipulada e usando uma quantidade x de vértices. A quantidade de vértices aumenta a cada iteração, sendo que x varia de 1 a n até que todos os focos

sejam alcançados (solução ótima). Em cada iteração, os focos computados são armazenados em `opt` e reaproveitados posteriormente, de maneira a reduzir os recálculos feitos pelo algoritmo de força bruta. Aqui, o Princípio da Otimalidade se aplica: para uma sequência ótima a partir de determinado vértice, cada subsequência computada também é ótima, i.e., contém o máximo de focos que podem ser alcançados com a restrição colocada ao número de vértices.

De maneira mais detalhada, pode-se descrever o processo de cálculo da solução via PD da seguinte forma:

- A matriz `opt` é inicializada na posição 1 com os focos relativos a cada vértice. Assim, `opt[1][1]` contém a sequência de focos do vértice 1 apenas e `opt[1][5]` contém a sequência de focos do vértice 5 apenas, por exemplo. A subestrutura ótima aqui considera o mínimo de vértices igual a 1 e, antes de iniciar o *loop while*, verifica se todos os focos podem ser alcançados com apenas um vértice.
- A cada iteração, são calculados os focos para um total de vértices igual a `min_vertices` a partir de cada vértice v . Esses valores são colocados na matriz `opt[min_vertices][v]`, a qual é utilizada para computar os focos posteriormente (sem necessidade de recálculo).

Um procedimento auxiliar para verificar se todos os focos foram alcançados (`CHECK-FOCUS`, na linha (10)) é apresentado no Algoritmo 3. Neste procedimento, verifica-se um vetor contendo todos os focos calculados para cada vértice em uma iteração do algoritmo, determinando se todos os focos foram alcançados para um mínimo de vértices igual a `min_vertices`.

De forma similar à força bruta, a busca em profundidade (`DEPTH-FIRST-SEARCH`) (linha 5) é feita apenas com a finalidade de obter uma direção única para verificar os vértices do grafo, o qual é cíclico.

Algoritmo 2: ZIKAZEROANELDUAL - Programação Dinâmica

Data: $G = (\mathbb{V}, \mathbb{A}, \mathbb{F}, \mathcal{R})$

Result: `best_vertices`, where `best_vertices` 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   best_vertices  $\leftarrow$  list containing all vertices
5   ordered_vertices  $\leftarrow$  DEPTH-FIRST-SEARCH( $\mathbb{V}, \mathbb{A}$ )
6   min_vertices  $\leftarrow 1$ 
7   Create opt as a multidimensional matrix of size  $n \times n \times r$ 
8   Initialize opt[min_vertices][v] with all the focuses from each vertex  $v \in \mathbb{V}$ 
9   min_vertices  $\leftarrow 2$ 
10  while CHECK-FOCUS(opt[min_vertices], min_vertices, best_vertices) does not find all focuses
    and min_vertices  $\leq n$  do
11    foreach  $v \in ordered\_vertices$  do
12       $next_v \leftarrow$  element in position  $(min\_vertices - 1)$  after  $v$  in ordered_vertices
13      opt[min_vertices][v] \leftarrow find common focuses between opt[min_vertices - 1][v] and
        opt[0][next_v]
14    end
15    min_vertices ++
16  end
17  return best_vertices
18 end

```

Algoritmo 3: ZIKAZEROANELDUAL - Programação Dinâmica - CHECK-FOCUS

Data: *focus_array, min_vertices, best_vertices*

Result: *found*, boolean that indicates if all focuses were found (true) or not (false)

```

1 begin
2   found ← false
3   for i in 1 to focus_array.length do
4     if focus_array[i] contains all focuses then
5       found ← true
6       vertices ← all vertices from i to i + min_vertices - 1
7       if sum of indexes in vertices is smaller than sum of indexes in best_vertices then
8         best_vertices ← vertices
9       end
10    end
11  end
12  return found
13 end

```

- **Algoritmo Guloso:** o pseudo-código para o paradigma guloso é apresentado no Algoritmo 4. O algoritmo basicamente segue os seguintes passos:

1. É escolhido um vértice inicial *best_element* com menor número de colisões com seus vizinhos, i.e., um vértice que contenha a menor quantidade de focos em comum com seus adjacentes. Se houver empate, escolhe o vértice de menor índice¹.
2. É escolhido, dentre os dois vértices adjacentes a *best_element*, aquele que obtiver a maior razão $rz_g = \frac{num_focos}{num_vertices}$, onde *num_focos* é o número de focos encontrado a partir do vértice adjacente explorando $num_vertices = x \leq v_{raio}$ vértices a partir dele. Essa razão é encontrada a partir da estratégia gulosa elaborada: para cada vértice adjacente, sua vizinhança é verificada até um raio de busca *vraio* pré-determinado. Se todos os focos forem encontrados antes que a investigação do raio de busca tenha terminado ou se o outro vértice adjacente pesquisado tenha sido encontrado, a busca pára e o *num_vertices* passa a ser a quantidade de vértices investigada até então. Dessa forma, a razão *rz_g* representa uma espécie de densidade de focos local, a qual deve ser maximizada em uma solução ótima.
3. Tendo escolhido o vértice adjacente com maior *rz_g*, ele é adicionado à solução ótima *best_vertices*, bem como seus focos de abrangência.
4. Se todos focos ainda não tiverem sido encontrados, repete-se os passos 2 a 4 para os vértices adjacentes ao conjunto *best_vertices*.

É importante ressaltar que os procedimentos CHECK-LEFT e CHECK-RIGHT são similares, sendo que o primeiro deles percorre o grafo em sentido horário e o segundo, anti-horário. Devido a esta semelhança, apenas o procedimento CHECK-LEFT é apresentado (Algoritmo 5). Tais procedimentos são utilizados para encontrar a razão *rz_g* do passo 2, conforme descrição acima.

¹Em todos os algoritmos implementados, o critério de desempate foi escolher a solução ótima com menor soma de vértices. Assim, o vértice de menor índice é escolhido no primeiro passo do algoritmo guloso de modo a minimizar uma possível soma do conjunto de vértices encontrado.

Algoritmo 4: ZIKAZEROANELDUAL - Algoritmo Guloso

Data: $G = (\mathbb{V}, \mathbb{A}, \mathbb{F}, \mathcal{R})$

Result: $best_vertices$, where $best_vertices$ 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      $ordered\_vertices \leftarrow \text{DEPTH-FIRST-SEARCH}(\mathbb{V}, \mathbb{A})$ 
5      $best\_vertices \leftarrow \text{empty list}$ 
6      $\mathbb{F}_{current} \leftarrow \text{empty list}$ 
    /* find best first element by minimizing collisions */
7      $best\_collisions \leftarrow r + 1$ 
8      $best\_element \leftarrow -1$ 
9     foreach  $v \in \mathbb{V}$  do
10         $collisions \leftarrow 0$ 
11        foreach  $v_{adj} \in \mathbb{A}(v)$  do
12             $collisions \leftarrow collisions + (\text{number of common focuses between } v \text{ and } v_{adj})$ 
13            if  $collisions < best\_collisions$  then
14                 $best\_collisions \leftarrow collisions$ 
15                 $best\_element \leftarrow v$ 
16            end
17        end
18    end
19    add  $best\_element$  to  $best\_vertices$ 
20    add focuses from  $best\_element$  to  $\mathbb{F}_{current}$ 
    /* check adjacent vertices */
21     $left_v \leftarrow \text{element in } ordered\_vertices \text{ in the left position to } best\_element$ 
22     $right_v \leftarrow \text{element in } ordered\_vertices \text{ in the right position to } best\_element$ 
23     $rzleft \leftarrow \text{CHECK-LEFT}(left_v, right_v, \mathbb{F}_{current})$ 
24     $rzright \leftarrow \text{CHECK-RIGHT}(right_v, left_v, \mathbb{F}_{current})$ 
25    while  $\mathbb{F}_{current}$  does not contain all focuses do
26        if  $rzleft > rzright$  then
27             $best\_element \leftarrow left_v$ 
28             $left_v \leftarrow \text{element in } ordered\_vertices \text{ in the left position to } best\_element$ 
29        else if  $rzleft < rzright$  then
30             $best\_element \leftarrow right_v$ 
31             $right_v \leftarrow \text{element in } ordered\_vertices \text{ in the right position to } best\_element$ 
32        else if  $left_v.index < right_v.index$  then
33             $best\_element \leftarrow left_v$ 
34             $left_v \leftarrow \text{element in } ordered\_vertices \text{ in the left position to } best\_element$ 
35        else
36             $best\_element \leftarrow right_v$ 
37             $right_v \leftarrow \text{element in } ordered\_vertices \text{ in the right position to } best\_element$ 
38        end
39        add  $best\_element$  to  $best\_vertices$ 
40        add focuses from  $best\_element$  to  $\mathbb{F}_{current}$ 
41         $rzleft \leftarrow \text{CHECK-LEFT}(left_v, right_v, \mathbb{F}_{current})$ 
42         $rzright \leftarrow \text{CHECK-RIGHT}(right_v, left_v, \mathbb{F}_{current})$ 
43    end
44    return  $best\_vertices$ 
45 end

```

Algoritmo 5: ZIKAZEROANELDUAL - Algoritmo Guloso - CHECK-LEFT

Data: vertex v , limiting vertex max_v , $current_focuses$

Result: rz

```

1 begin
2   |  $focuses_v \leftarrow$  (focuses accessed from  $v$ ) +  $current\_focuses$ 
3   |  $num\_vertices \leftarrow 1$ 
4   |  $v_{raio} \leftarrow 3$  //  $v_{raio}$  can have any value from 1 to  $n$ 
5
6   | while  $focuses_v$  does not contain all focuses and  $num\_vertices < v_{raio}$  and  $max_v$  was not
      reached do
7     |   | add the focuses from the next vertex to  $focuses_v$ 
8     |   |  $num\_vertices \leftarrow num\_vertices + 1$ 
9   | end
10  |  $rz \leftarrow \frac{focuses_v}{num\_vertices}$ 
11  | return  $rz$ 
12 end

```

Exercício 2

Analise as complexidades temporais e espaciais (usando Notação Assintótica) dos algoritmos propostos no Exercício 1.

Conforme especificado, o grafo é cíclico, não-orientado, cada um de seus vértices se liga a apenas outros dois ($|E| = |V|$) e cada vértice acessa somente 2 focos. Por se tratar de grafo com tal configuração, todos os algoritmos utilizam uma matriz de adjacência e uma matriz de focos, ambas de tamanho $2n$, onde $n = |V|$. Assim sendo, além de suas complexidades espaciais específicas, todos os algoritmos adicionam o custo espacial de tais estruturas, o qual totaliza $4n$.

Para a análise da complexidade temporal, serão consideradas custosas as operações de adição e atribuição em vetores, bem como comparações realizadas nos mesmos.

1. Busca por Força Bruta(a) *Complexidade Espacial*

Ao observar o Algoritmo 1, pode-se verificar que ele utiliza dois vetores auxiliares, ambos de tamanho n : $best_vertices$ e $ordered_vertices$. Além disso, cada iteração também utiliza os vetores $\mathbb{V}_{current}$, de tamanho máximo n , e $\mathbb{F}_{current}$, de tamanho máximo r . Dessa forma, a complexidade espacial, adicionada às matrizes de adjacência e focos, é: $f(n, r) = 4n + 2n + n + r = \mathcal{O}(n+r)$.

(b) *Complexidade Temporal*

Ao observar o Algoritmo 1, as seguintes operações e respectivos custos podem ser verificados:

- A DEPTH-FIRST-SEARCH possui custo de $\mathcal{O}(n+m) = \mathcal{O}(n)$, uma vez que $m = n$.
- Para cada vértice v , o loop while das linhas (9) a (13) verifica se o vetor $\mathbb{F}_{current}$ não possui todos os focos e, caso não possua, adiciona o próximo elemento que se segue a v e seu respectivo conjunto de focos. Dessa forma, no máximo $n - 1$ vértices serão adicionados a $\mathbb{V}_{current}$ no pior caso², bem como serão feitas, no máximo, $n - 1$ verificações de focos em $\mathbb{F}_{current}$ a um custo de r . Assim, a cada iteração, um elemento é adicionado a $\mathbb{V}_{current}$ e seus dois focos são adicionados a $\mathbb{F}_{current}$, bem como $\mathbb{F}_{current}$ é verificado quanto à quantidade de focos já abrangida. Como são realizadas no máximo $n - 1$ iterações, o loop while totaliza, portanto,

²O pior caso ocorre quando todos os focos só podem ser alcançados com todos os vértices do grafo.

um custo temporal de $(n - 1) \cdot (r + 1 + 2) = \mathcal{O}(nr)$. Para os n vértices verificados, temos um custo total de $\mathcal{O}(n^2r)$.

- Para cada vértice v , também é feita uma comparação nas linhas (14) e (16) entre o vetor $\mathbb{V}_{current}$ e $best_vertices$. Tal operação possui custo n para cada iteração do while. Para os n vértices, temos um custo total de $\mathcal{O}(n^2)$.

Tendo em vista tais considerações, pode-se dizer que a complexidade temporal do Algoritmo de Força Bruta proposto é $\mathcal{O}(n) + \mathcal{O}(n^2r) + \mathcal{O}(n^2) = \mathcal{O}(n^2r)$ (pior caso). No melhor caso, quando todos os vértices contivessem todos os focos, o while realizaria apenas uma iteração, tendo complexidade de $\mathcal{O}(nr)$. Assim, a complexidade temporal total do algoritmo de força bruta seria de $\mathcal{O}(n) + \mathcal{O}(nr) + \mathcal{O}(n^2) = \mathcal{O}(nr + n^2)$ no melhor caso.

2. Programação Dinâmica

(a) Complexidade Espacial

Ao observar o Algoritmo 2, pode-se verificar que ele utiliza dois vetores auxiliares, ambos de tamanho n : $best_vertices$ e $ordered_vertices$. Além disso, a matriz opt possui tamanho $n \times n \times r$, possuindo um custo de $\mathcal{O}(n^2r)$. Assim, o custo espacial total pode ser calculado como $f(n, r) = 4n + 2n + n^2r = \mathcal{O}(n^2r)$.

(b) Complexidade Temporal

Ao observar o Algoritmo 2, as seguintes operações e respectivos custos podem ser verificados:

- A DEPTH-FIRST-SEARCH possui custo de $\mathcal{O}(n + m) = \mathcal{O}(n)$, uma vez que $m = n$.
- A inicialização da matriz opt na linha (8) possui custo $2n = \mathcal{O}(n)$.
- Para cada vértice v , o loop while das linhas (10) a (16) faz uma operação OR bit a bit entre o valor prévio de $opt[min_vertices - 1][v]$ e os focos do elemento que está ($min_vertices - 1$) posições após v . Essa operação possui custo temporal r para cada vértice. Logo, o custo para todos os vértices é $\mathcal{O}(nr)$.
- Em cada iteração, o procedimento CHECK-FOCUS verifica se todos os focos foram alcançados, tendo um custo de $\mathcal{O}(nr)$, uma vez que são verificados todos os focos para cada vértice v .
- Ainda em CHECK-FOCUS, a atribuição de vértices a $best_vertices$ é realizada apenas na última iteração feita pelo algoritmo, quando todos os focos são encontrados em algum ponto. No pior caso, seu custo é $\mathcal{O}(n^2)$, considerando-se que cada vértice alcançará todos os focos e a atribuição será feita a cada vértice $v \in \mathbb{V}$.
- O pior caso para o algoritmo se encontra na situação em que o mínimo de vértices que acessa todos os focos é igual ao conjunto de todos os vértices do grafo. Nesse caso, são realizadas n iterações.

Tendo em vista tais considerações, pode-se dizer que a complexidade temporal do Algoritmo de Programação Dinâmica proposto é $\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n^2r) + \mathcal{O}(n^2r) = \mathcal{O}(n^2r)$ (pior caso). No melhor caso, quando pelo menos um vértice contém todos os focos, sua complexidade temporal é de $\mathcal{O}(nr)$.

3. Algoritmo Guloso

(a) Complexidade Espacial

Ao observar o Algoritmo 4, pode-se verificar que ele utiliza dois vetores auxiliares, ambos de tamanho n : $best_vertices$ e $ordered_vertices$. Além disso, também utiliza os vetores $\mathbb{F}_{current}$ e $focuses_v$ (nos procedimentos CHECK-LEFT e CHECK-RIGHT) de tamanho máximo r . Assim sendo, sua complexidade espacial total, adicionada ao custo das matrizes de adjacência e focos, é de $4n + 2n + 2r = \mathcal{O}(n + r)$.

(b) *Complexidade Temporal*

Ao observar o Algoritmo 4, as seguintes operações e respectivos custos podem ser verificados:

- A DEPTH-FIRST-SEARCH possui custo de $\mathcal{O}(n + m) = \mathcal{O}(n)$, uma vez que $m = n$.
- Para cada vértice v , o *loop for* das linhas (9) a (18) calcula o número de "colisões" (i.e., focos em comum) entre v e seus dois vértices adjacentes e obtém o vértice que minimiza as "colisões". Como o custo de calcular os focos em comum é r , a complexidade temporal total do *loop for* é de $\mathcal{O}(nr)$.
- Os procedimentos CHECK-LEFT e CHECK-RIGHT (Algoritmo 5) dependem do valor do raio de pesquisa colocado (v_{raio}), além do número de focos. Como a adição de focos a $focuses_v$ possui custo constante (igual a 2), a operação custosa envolve o a verificação do número total de focos em $focuses_v$, a qual é feita com custo igual a r a cada iteração do *while*. No pior caso, são realizadas v_{raio} iterações, quando não são encontrados todos os focos até o limite de busca. Assim, a complexidade temporal total de CHECK-LEFT e CHECK-RIGHT é de $\mathcal{O}(rv_{raio})$. Como foi considerado $v_{raio} = 3$ para todas as execuções do atual cenário, a complexidade é de $3r = \mathcal{O}(r)$, em nosso caso³.
- O *loop while* das linhas (25) a (43) permite que os procedimentos CHECK-LEFT e CHECK-RIGHT sejam continuamente chamados até que todos os focos tenham sido encontrados. Cada iteração possui um custo temporal de $2\mathcal{O}(r) = \mathcal{O}(r)$, uma vez que cada procedimento é chamado duas vezes. No pior caso, quando todos os vértices formam a solução, são realizadas $n - 1$ iterações, de modo que a complexidade temporal total é de $(n - 1) \cdot \mathcal{O}(r) = \mathcal{O}(nr)$.

Tendo em vista tais considerações, pode-se dizer que a complexidade temporal do Algoritmo Guloso proposto é $\mathcal{O}(n) + \mathcal{O}(nr) + \mathcal{O}(nr) = \mathcal{O}(nr)$ (pior caso). No melhor caso, quando todos os vértices contivessem todos os focos, o *while* não realizaria iteração. Assim, a complexidade temporal total do algoritmo se manteria como $\mathcal{O}(nr)$, uma vez que o algoritmo é limitado pela operação de busca do melhor primeiro elemento, a qual possui complexidade $\mathcal{O}(nr)$ para todos os casos.

Exercício 3

Implemente os algoritmos propostos no Exercício 1 na linguagem de programação C, C++, JAVA ou PYTHON.

A implementação foi feita em PYTHON e se encontra na pasta "codigo" do presente trabalho. O código dos algoritmos de força bruta, programação dinâmica e guloso se encontram, respectivamente, nas pastas "bruto", "dinamico" e "guloso".

Exercício 4

Execute testes da implementação do Exercício 3, propondo instâncias interessantes para o Problema ZikaZeroAnelDual.

Foram realizados testes para as seguintes instâncias⁴:

³É importante ressaltar que, quanto maior o valor de v_{raio} , mais próxima estará a solução do guloso de um valor ótimo. Isso ocorre porque o raio de busca aumenta e, consequentemente, a certeza de que o vértice obtido é o candidato mais adequado para se adicionar à solução ótima sendo construída. O valor $v_{raio} = 3$ foi arbitrariamente escolhido para o presente problema, porém é necessário um estudo mais detalhado para determinação de um valor adequado para este parâmetro que minimize o impacto ao custo total do algoritmo.

⁴Todas as instâncias utilizadas nos testes se encontram na pasta "tests" do presente trabalho.

- **Conjunto 1:** conjunto de dados que constituem o melhor caso dos algoritmos. O número de focos foi mantido em 2 (apenas dois focos a serem acessados), variando-se o número de vértices de 50 a 950, com incremento de 50 ($|V| = \{50, 100, 150, \dots, 950\}$). Nesse caso, qualquer um dos vértices é solução do algoritmo.
- **Conjunto 2:** nesse conjunto, o número de focos foi mantido em 20, variando-se o número de vértices de 50 a 950, com incremento de 50 ($|V| = \{50, 100, 150, \dots, 950\}$).
- **Conjunto 3:** nesse conjunto, o número de vértices foi mantido em 600, variando-se o número de focos de 50 a 500, com incremento de 50 ($|V| = \{50, 100, 150, \dots, 500\}$).
- **Conjunto 4:** conjunto de dados que constituem o pior caso dos algoritmos. Nesse conjunto, variou-se o número de vértices de 50 a 600, com incremento de 50 ($|V| = \{50, 100, 150, \dots, 600\}$). O número de focos, por sua vez, correspondeu ao número de vértices acrescido de 1, de modo a gerar instâncias com $r = n + 1$ cada (assim, na instância com 50 vértices, havia 51 focos, por exemplo). Nesse caso, os conjuntos foram gerados de modo que a solução é o conjunto de todos os vértices do grafo.

A seguir são apresentados os resultados dos testes, contendo suas tabelas e respectivos gráficos de representação. É importante salientar que as linhas dos gráficos correspondem à função polinomial que melhor se ajusta aos dados.

1. Testes do Conjunto 1

Para os testes do Conjunto 1 (melhor caso), o resultado da execução do algoritmo é apresentado na Tabela 1. O gráfico que apresenta a relação entre o tempo de execução e o número de vértices é apresentado na Figura 1.

n	Bruto (s)	Dinâmico (s)	Guloso (s)
50	0.119	0.120	0.115
100	0.124	0.123	0.116
150	0.125	0.120	0.137
200	0.132	0.124	0.124
250	0.153	0.130	0.126
300	0.163	0.129	0.128
350	0.178	0.133	0.130
400	0.199	0.138	0.134
450	0.223	0.143	0.142
500	0.246	0.140	0.144
550	0.281	0.150	0.133
600	0.300	0.151	0.151
650	0.325	0.157	0.146
700	0.358	0.159	0.153
750	0.386	0.171	0.156
800	0.422	0.174	0.159
850	0.445	0.173	0.163
900	0.490	0.181	0.170
950	0.533	0.184	0.170

Tabela 1: Relação entre o Número de Vértices do Grafo e o Tempo de Execução dos Algoritmos para o Conjunto 1

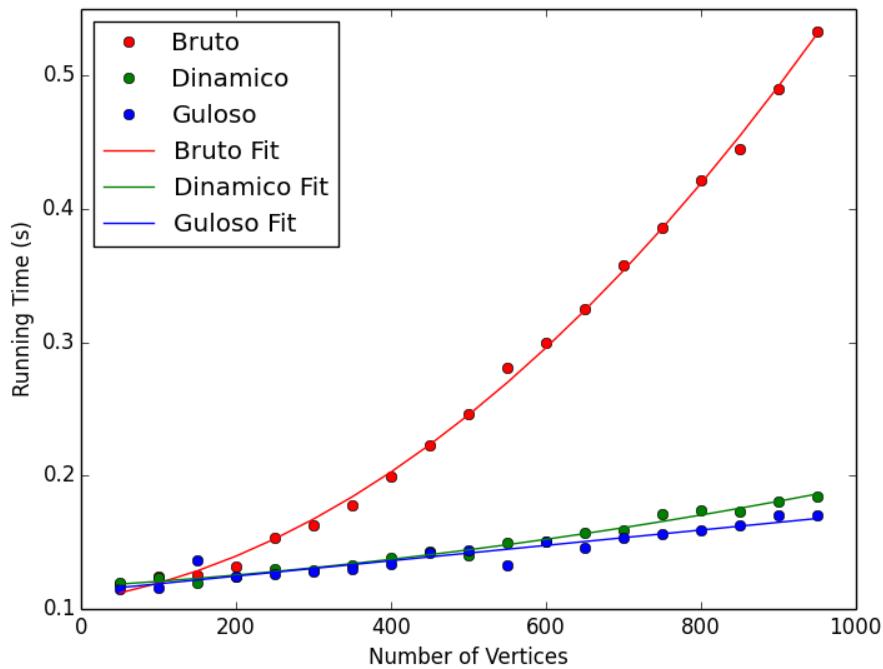


Figura 1: Relação entre Tempo de Execução e Número de Vértices - Conjunto 1 (Melhor Caso)

As linhas correspondem à função polinomial que melhor se ajusta aos dados.

2. Testes do Conjunto 2

Para os testes do Conjunto 2, o resultado da execução do algoritmo é apresentado na Tabela 2. O gráfico que apresenta a relação entre o tempo de execução e o número de vértices é apresentado na Figura 2.

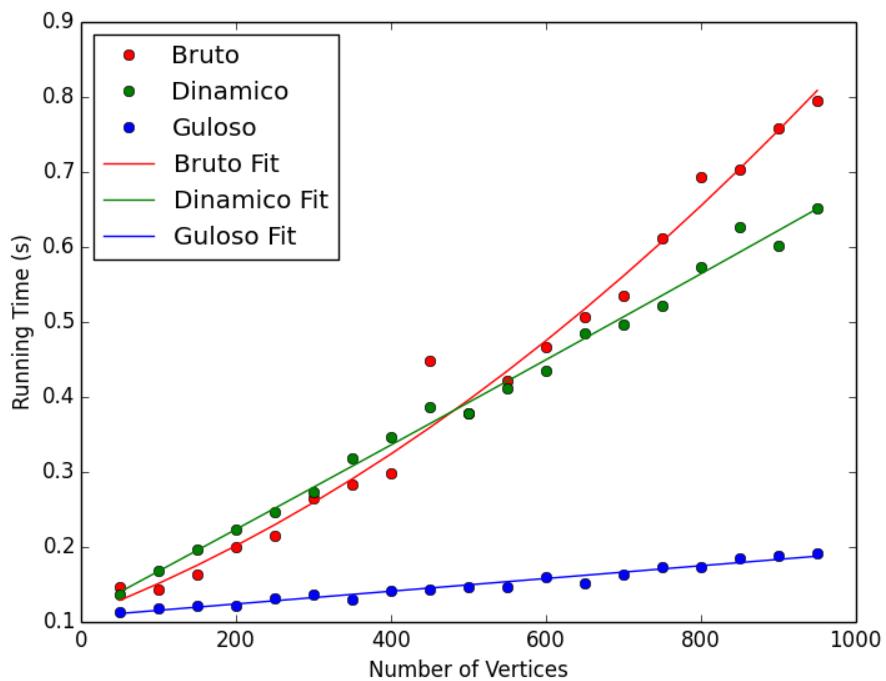
3. Testes do Conjunto 3

Para os testes do Conjunto 3, o resultado da execução do algoritmo é apresentado na Tabela 3. O gráfico que apresenta a relação entre o tempo de execução e o número de vértices é apresentado na Figura 3. Devido à escala impossibilitar a visualização do algoritmo guloso, foi gerada um novo gráfico (Figura 4) contendo apenas os dados referentes ao mesmo.

4. Testes do Conjunto 4

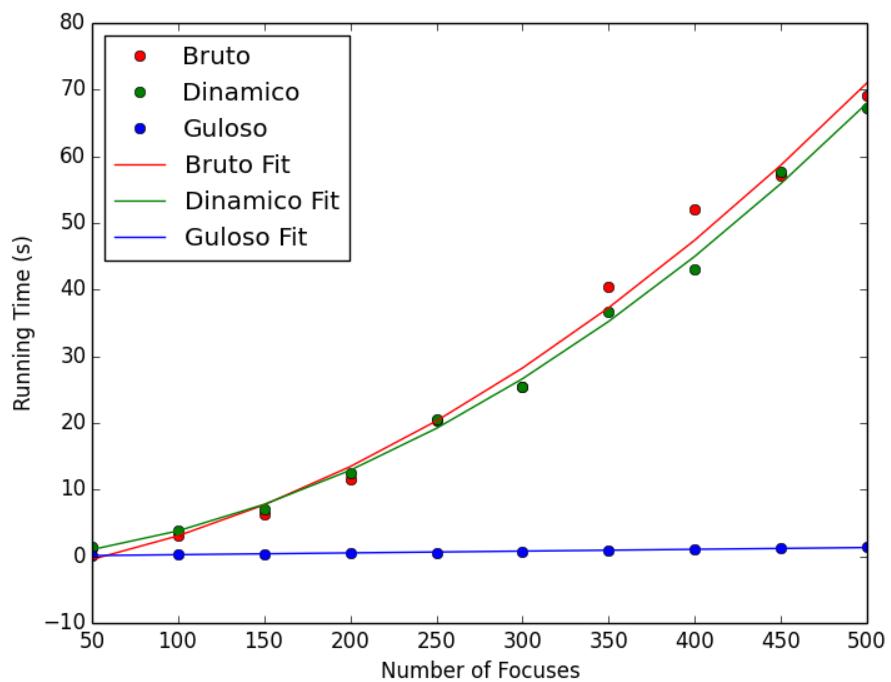
Para os testes do Conjunto 3, o resultado da execução do algoritmo é apresentado na Tabela 4. O gráfico que apresenta a relação entre o tempo de execução e o número de vértices é apresentado na Figura 5. Dado que o número de focos e vértices variam de forma linear ($r = n + 1$), pode-se considerar o eixo das abscissas pode ser aproximado para considerar a variação do número de vértices ou focos. Devido à escala impossibilitar a visualização do algoritmo guloso, foi gerada um novo gráfico (Figura 6) contendo apenas os dados referentes ao mesmo.

n	Bruto (s)	Dinâmico (s)	Guloso (s)
50	0.147	0.137	0.114
100	0.144	0.168	0.119
150	0.164	0.196	0.122
200	0.200	0.224	0.122
250	0.215	0.247	0.132
300	0.265	0.274	0.136
350	0.284	0.318	0.130
400	0.299	0.346	0.142
450	0.448	0.387	0.144
500	0.378	0.379	0.147
550	0.421	0.412	0.146
600	0.467	0.435	0.160
650	0.506	0.485	0.152
700	0.535	0.497	0.163
750	0.612	0.522	0.174
800	0.694	0.574	0.174
850	0.704	0.627	0.185
900	0.758	0.601	0.188
950	0.795	0.652	0.191

Tabela 2: Relação entre o Número de Vértices do Grafo e o Tempo de Execução dos Algoritmos para o Conjunto 2**Figura 2:** Relação entre Tempo de Execução e Número de Vértices - Conjunto 2

As linhas correspondem à função polinomial que melhor se ajusta aos dados.

r	Bruto (s)	Dinâmico (s)	Guloso (s)
50	1.172	1.349	0.200
100	3.099	3.878	0.320
150	6.360	7.011	0.355
200	11.537	12.464	0.432
250	20.409	20.567	0.571
300	25.375	25.531	0.766
350	40.469	36.668	0.832
400	52.056	43.091	1.064
450	57.104	57.707	1.239
500	69.195	67.226	1.438

Tabela 3: Relação entre o Número de Focos do Grafo e o Tempo de Execução dos Algoritmos para o Conjunto 3**Figura 3:** Relação entre Tempo de Execução e Número de Focos - Conjunto 3

As linhas correspondem à função polinomial que melhor se ajusta aos dados.

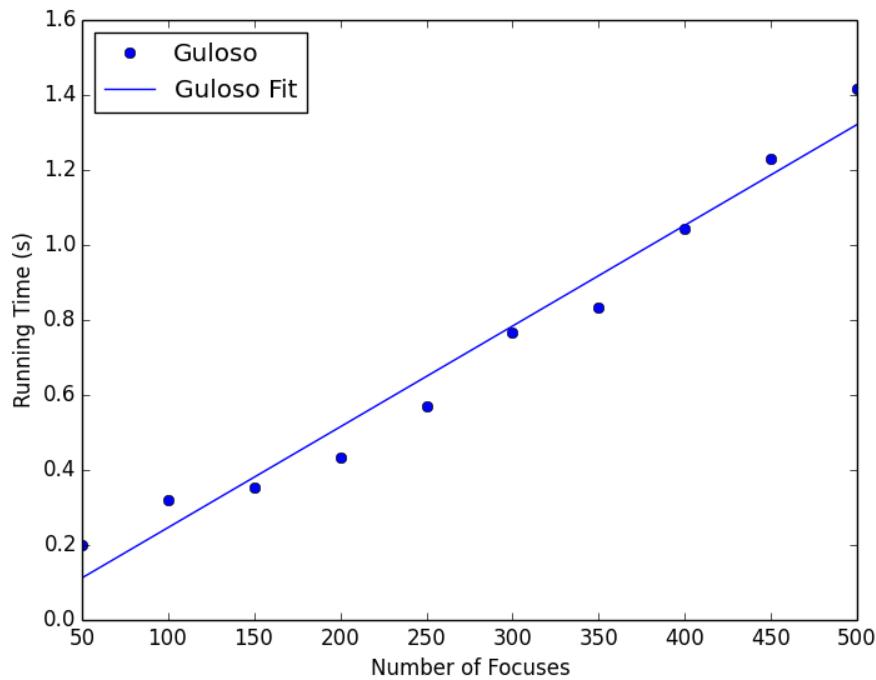
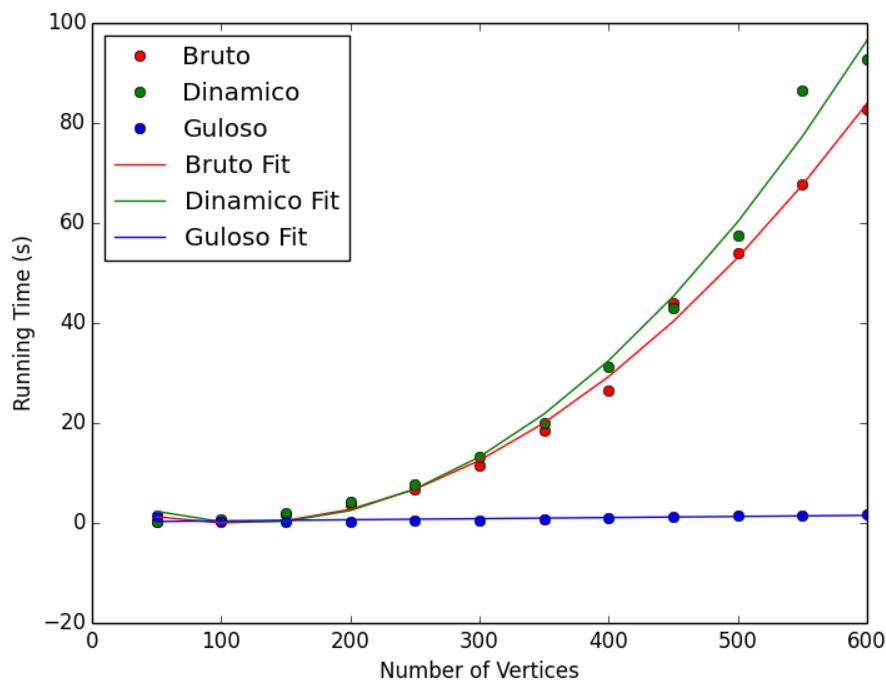


Figura 4: Relação entre Tempo de Execução e Número de Focos - Conjunto 3 - Algoritmo Guloso

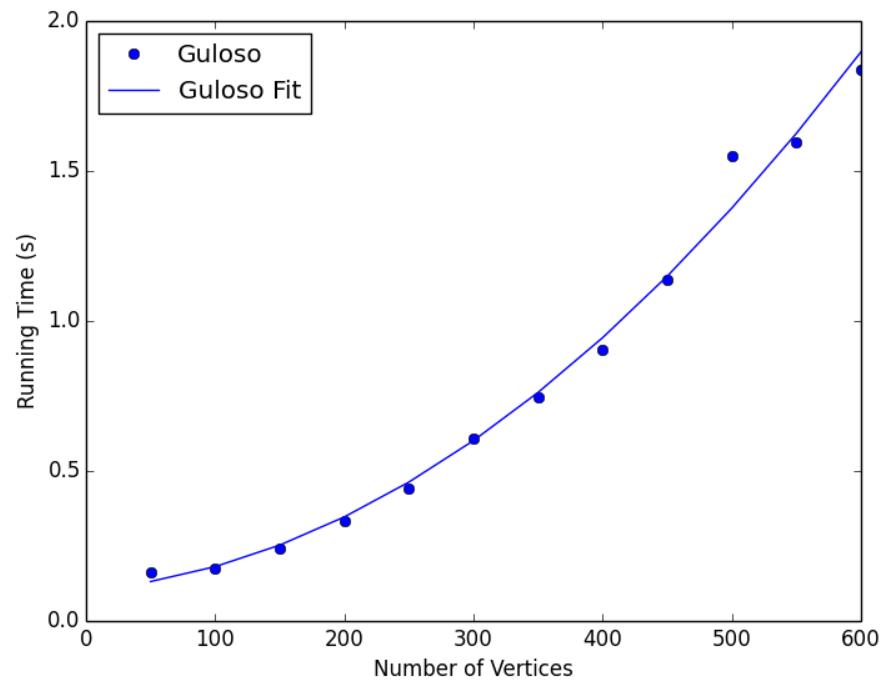
A linha corresponde à função polinomial que melhor se ajusta aos dados.

n / r	Bruto (s)	Dinâmico (s)	Guloso (s)
50	0.191	0.235	0.161
100	0.590	0.718	0.174
150	1.717	1.974	0.243
200	3.628	4.338	0.335
250	6.633	7.695	0.441
300	11.397	13.261	0.610
350	18.471	20.039	0.744
400	26.624	31.193	0.904
450	43.960	42.909	1.138
500	54.061	57.486	1.551
550	67.745	86.513	1.594
600	82.843	92.762	1.836

Tabela 4: Relação entre o Número de Vértices / Focos do Grafo e o Tempo de Execução dos Algoritmos para o Conjunto 4

**Figura 5:** Relação entre Tempo de Execução e Número de Vértices / Focos - Conjunto 4

As linhas correspondem à função polinomial que melhor se ajusta aos dados.

**Figura 6:** Relação entre Tempo de Execução e Número de Vértices / Focos - Conjunto 4 - Algoritmo Guloso

As linhas correspondem à função polinomial que melhor se ajusta aos dados.

Exercício 5

Compare a análise e a execução, respectivamente, Exercícios 2 e 4. Principalmente, relate se as previsões teóricas estão em concordância com os resultados experimentais.

- **Conjunto 1:**

Para os testes do Conjunto 1 (melhor caso), percebe-se que todos os algoritmos apresentaram valores similares de tempo de execução para a menor instância (50 vértices). Dentre os algoritmos, percebe-se que o algoritmo de Força Bruta apresentou a maior variação de tempo, registrando 0.119 segundos para a menor instância e 0.533 para a maior. Isto significa um aumento de cerca de 4.5 vezes no tempo de execução da menor para a maior instância. Os algoritmos de Programação Dinâmica e Guloso, por outro lado, registraram uma variação de tempo de execução menos acentuada: o dinâmico e o guloso registraram um aumento de cerca de 1.5 vezes entre a maior e menor instância. Tais constatações podem também ser observadas na Figura 1. Na figura, nota-se que a curva que melhor se ajusta aos pontos indica que o algoritmo de Força Bruta cresce de forma não-linear com o aumento de n , enquanto o crescimento parece linear para os algoritmos de programação dinâmica e guloso.

Tal resultado está em acordo com a complexidade temporal verificada no melhor caso para os algoritmos no Exercício 2. Para o melhor caso, o algoritmo de Força Bruta possui complexidade $\mathcal{O}(nr+n^2)$. Como o número de focos é fixado em 2 para o Conjunto 1 ($r = 2$), o custo passa a ser $\mathcal{O}(2n + n^2) = \mathcal{O}(n^2)$, i.e., quadrático em relação ao número de vértices. Para os algoritmos de programação dinâmica e guloso, por outro lado, o custo do melhor caso é $\mathcal{O}(nr)$, conforme Exercício 2. No cenário do conjunto 1 (número de focos igual a 2), a complexidade é de $\mathcal{O}(2n) = \mathcal{O}(n)$, i.e., linear para o tamanho da entrada. Tal resultado está em conformidade com o que mostra o gráfico para os algoritmos de programação dinâmica e guloso.

- **Conjunto 2:**

Para os testes do Conjunto 2 (número de focos fixo em 20), percebe-se que os algoritmos de força bruta e programação dinâmica apresentaram maior variação quanto ao tempo de execução entre a menor e a maior instância, sendo que o primeiro apresentou um desempenho mais baixo para instâncias maiores que 500. A variação do tempo de execução foi acentuada para esses dois algoritmos: o algoritmo de força bruta demorou cerca de 5.4 vezes mais para executar a maior instância em comparação à menor; o algoritmo dinâmico, por sua vez, demorou cerca de 4.8 vezes mais. Diferentemente dos anteriores, o algoritmo guloso apresentou uma variação bem menor, sendo que a maior instância levou 1.7 vezes mais tempo do que a menor instância. Tais constatações podem ser visualizadas no gráfico da Figura 2. Na figura, percebe-se que o algoritmo guloso apresenta um crescimento similar a uma função linear de baixo coeficiente angular (reta possui baixa inclinação). Por outro lado, os algoritmos de programação dinâmica e força bruta apresentaram maior inclinação.

Tal resultado está de acordo com a complexidade temporal verificada para o algoritmo guloso no Exercício 2. Como a complexidade temporal do algoritmo guloso foi calculada em $\mathcal{O}(nr)$ para todos os casos, o custo para o conjunto 2 seria de $\mathcal{O}(20n) = \mathcal{O}(n)$, uma vez que o número de focos é fixo em 20. Como pode-se observar no gráfico da Figura 2, o algoritmo guloso exibiu um crescimento linear, conforme esperado. Os algoritmos de força bruta e programação dinâmica, por sua vez, possuem ambos pior caso de $\mathcal{O}(n^2r)$, o que seria igual a $\mathcal{O}(20n^2) = \mathcal{O}(n^2)$ para o conjunto 2 ($r = 20$). Apesar da curva ajustada ao algoritmo de força bruta ser similar à de uma função quadrática, o algoritmo de programação dinâmica apresentou um crescimento similar ao linear. Tal comportamento do algoritmo PD pode ser justificado pelo conjunto 2 ser considerado um cenário intermediário entre o melhor e o pior caso: como o melhor caso do PD é $\mathcal{O}(20n) = \mathcal{O}(n)$ para o conjunto 2, sua complexidade pode estar em um intermediário entre o melhor e o pior caso. Para o algoritmo de força bruta, o melhor

caso também é quadrático com o tamanho da entrada, de modo que o comportamento exibido foi semelhante ao esperado.

- **Conjunto 3:**

Para os testes do Conjunto 3 (número de vértices fixo em 600), percebe-se que os algoritmos de força bruta e programação dinâmica apresentaram maior variação quanto ao tempo de execução entre a menor e a maior instância, tendo ambos resultados muito semelhantes. A variação do tempo de execução foi acentuada para esses dois algoritmos: o algoritmo de força bruta demorou cerca de 59.0 vezes mais para executar a maior instância em comparação à menor; o algoritmo dinâmico, por sua vez, demorou cerca de 49.8 vezes mais. Diferentemente dos anteriores, o algoritmo guloso apresentou uma variação bem menor, sendo que a maior instância levou 7.2 vezes mais tempo do que a menor instância. Tais constatações podem ser visualizadas no gráfico da Figura 3. Apesar de haver variação nos valores pelo algoritmo guloso (conforme visto na Tabela 3), não foi possível visualizar com clareza a curva do mesmo na figura devido à escala do gráfico. Em virtude disso, seus resultados podem melhor visualizados na Figura 4.

Tal resultado está de acordo com a complexidade temporal verificada para o algoritmo guloso no Exercício 2. Como a complexidade temporal do algoritmo guloso foi calculada em $\mathcal{O}(nr)$ para todos os casos, o custo para o conjunto 3 seria de $\mathcal{O}(600r) = \mathcal{O}(r)$, uma vez que o número de vértices é fixo em 600. Como pode-se observar no gráfico da Figura 4, o algoritmo guloso exibiu um crescimento linear com relação ao número de focos, conforme esperado. Os algoritmos de força bruta e programação dinâmica, por sua vez, possuem ambos pior caso de $\mathcal{O}(n^2r)$, devendo possuir comportamento linear com relação ao número de focos. Tal cenário, no entanto, não se refletiu na curva ajustada para os resultados experimentais. Nesse caso, talvez seria interessante realizar mais experimentos para melhor investigar o comportamento dos algoritmos para o cenário com variação de focos e número fixo de vértices.

- **Conjunto 4:**

Para os testes do Conjunto 4 (pior caso), percebe-se que os algoritmos de força bruta e programação dinâmica apresentaram maior variação quanto ao tempo de execução entre a menor e a maior instância, tendo ambos resultados muito semelhantes. Diferentemente dos anteriores, o algoritmo guloso apresentou uma variação bem menor, sendo que a maior instância levou 11.4 vezes mais tempo do que a menor instância. Tais constatações podem ser visualizadas no gráfico da Figura 5. Apesar de haver variação nos valores pelo algoritmo guloso (conforme visto na Tabela 4), não foi possível visualizar com clareza a curva do mesmo na figura devido à escala do gráfico. Em virtude disso, seus resultados podem melhor visualizados na Figura 6.

Tal resultado está de acordo com a complexidade temporal verificada para o algoritmo guloso no Exercício 2. Como a complexidade temporal do algoritmo guloso foi calculada em $\mathcal{O}(nr)$ para todos os casos, o custo para o conjunto 4 seria de $\mathcal{O}(n \cdot (n + 1)) = \mathcal{O}(n^2)$, uma vez que o número de vértices e focos varia de modo que $r = n + 1$. Como pode-se observar no gráfico da Figura 6, o algoritmo guloso exibiu um crescimento similar a uma curva com relação ao número de vértices, conforme esperado. Os algoritmos de força bruta e programação dinâmica, por sua vez, possuem ambos pior caso de $\mathcal{O}(n^2r)$ e possuiriam custo de $\mathcal{O}(n^3)$ para o conjunto 4, visto que $r = n + 1$. Tendo em vista os resultados experimentais, a curva ajustada aos pontos desses algoritmos pode se assemelhar à previsão teórica feita, porém é necessário um refinamento dos experimentos para corroborar tais observações.

Análise Geral: Os algoritmos de programação dinâmica e força bruta apresentaram maior tempo de execução para a maioria dos experimentos, sendo que o primeiro apresentou melhor desempenho no melhor caso, no qual está em conformidade com sua previsão teórica. Conforme esperado, o algoritmo guloso apresentou desempenho bastante superior aos demais para todos os conjuntos de teste, seguindo também a

complexidade temporal calculada previamente. Apesar disso, o guloso encontra uma solução aproximada à ótima na maioria das vezes e, em alguns casos, nem mesmo consegue se aproximar à solução ótima.

A partir disso, os experimentos foram de grande valia para melhor compreensão de cada paradigma, mostrando a necessidade de se balancear desempenho e otimalidade de acordo com o cenário a ser avaliado. Em alguns casos, pode-se optar por prezar pelo desempenho (com um algoritmo guloso, por exemplo), sem garantir a otimalidade. Em determinadas situações, no entanto, é necessário abrir mão do desempenho para que a solução ótima possa ser encontrada, devendo-se recorrer a algoritmos mais custosos.

Trabalho Prático de Projeto e Análise de Algoritmos

Paradigmas de Projeto de Algoritmos - ZicaZeroAnelDual

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 **ZikaZeroAnelDual**, que consiste em, a partir de um conjunto de voluntários que tenham 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. Esse conjunto de voluntários é formado através de uma corrente do bem. Inicialmente, há um primeiro voluntário, que indica um amigo, que por sua vez indica outro, e assim sucessivamente, até que o último indica o primeiro (fechando um ciclo). Além disso, nessa campanha, cada voluntário deve visitar a exatamente dois criadouros.

O problema pode ser formalizado da seguinte forma: dados um grafo anel (aquele em que todo vértice conecta-se a exatamente outros dois) $\mathcal{G}_o(V, A)$, em que V é o conjunto de n voluntários e A é o conjunto de $m = |V|$ laços de amizade, um conjunto F dos r focos de reprodução do mosquito, e, uma relação $\mathcal{R}(v) : V \rightarrow F$, definida para cada $v \in V$, explicitando os focos de reprodução aos quais o voluntário v tem acesso, sendo que $|\mathcal{R}(v)| = 2, \forall v \in V$. 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 \mathcal{G}_o é conexo.

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 fim de facilitar o entendimento, a seção de modelagem foi dividida em duas partes. A primeira é destinada a explicar a estrutura de dados utilizada para representar os dados em todos os tipos de abordagens de solução propostos. A segunda é destinada a explicação dos algoritmos utilizando algoritmos

com abordagens baseadas em **Força Bruta**, **Programação Dinâmica** e **Gulosa** [Cormen 2002][Gross and Yellen 2004][Shuai and Hu 2006].

2.1. Modelagem dos Dados

Dado que o problema é formalizado e modelado como um grafo, e também contém restrições específicas a quantidade de focos acessado por cada voluntário, junto a quantidade de ligações de amizade entre os mesmos, utilizaremos uma estrutura um pouco diferente das convencionais utilizadas na literatura. Analizando a estrutura do anel, percebemos que cada vértice contém somente duas arestas e dois focos de acesso. Logo, podemos desenvolver uma estrutura de dados de tamanho fixo para armazenar cada vértice e suas relações. A figura 1 representa a estrutura de dados utilizada.



Figura 1. Exemplo da estrutura de dados utilizada para representar um vértice e suas relações

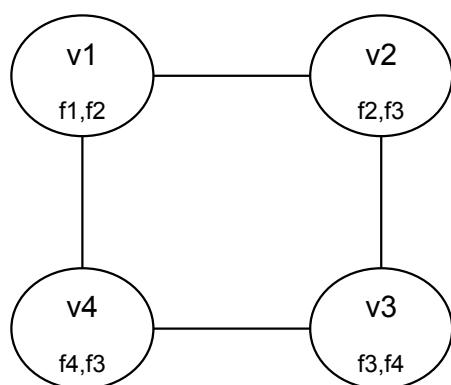


Figura 2. Exemplo de um grafo em forma de anel utilizado como entrada do problema

A estrutura de dados utiliza uma palavra para armazenar o índice do vértice, duas para armazenar os dois índices dos focos cobertos, uma para o vértice vizinho a direita (próximo vértice no sentido horário) e uma para a esquerda (sentido anti-horário). Assim sendo, cada vértice necessita de uma quantidade fixa de palavras (no nosso caso é 5) para ser armazenado. Além do mais, a estrutura apresentada facilita a recuperação dos dados de focos armazenados, sendo seu acesso direto. É utilizado um vetor contínuo para armazenar as estruturas dos vértices, em que os índices dos nós correspondem a posição

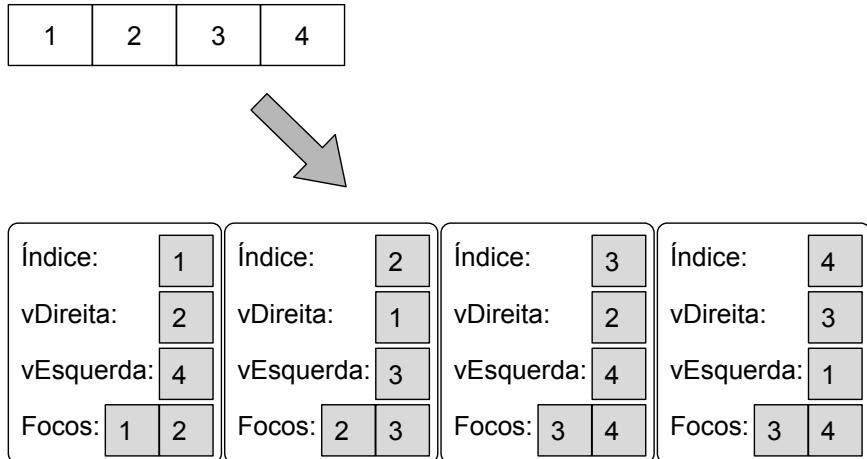


Figura 3. Exemplo da representação do grafo dado na figura 2

no vetor. Assim, o acesso a um vértice também é direto, sendo seu índice conhecido. A figura 3 exemplifica a representação dos dados relativos ao grafo de exemplo apresentado na figura 2.

2.2. Modelagem dos Algoritmos

2.2.1. Força Bruta

Para resolver o problema **ZikaZeroAnelDual** de através de uma abordagem de **Força Bruta**, utilizamos algumas restrições impostas na definição do problema para impedir que o custo de busca no espaço de soluções seja exponencial. Considerando que uma solução deve ser conexa, e em um grafo do tipo anel somente é possível obter subgrafos conexos que estejam em uma sequência no anel, então avaliamos todos os subconjuntos de vértices em sequência para ver o que leva a melhor solução. A figura 4 exemplifica o conjunto de possíveis soluções.

Para gerar o espaço de amostra, consideramos que um vértice qualquer possa pertencer a solução. Além disso consideramos que ele pode conter de um até $|V|$ elementos. Então, para cada vértice, consideramos que o mesmo seja o vértice mais a esquerda da solução, e caminhamos para a direita adicionando possíveis elementos a solução até que ela tenha o tamanho desejado. Assim, avaliamos todos os vértices sendo o mais a direita da solução, e para cada um é gerado soluções com tamanhos variando de um a $|V|$ elementos. Abaixo é apresentado subgrafos a serem analisados a partir da estratégia apresentada, com um caminhamento da esquerda para a direita no grafo. Sempre que uma folha é encontrada os vértices de seu caminho são retornados como um subconjunto conexo.

Subconjuntos conexos:

- | | |
|-----------------------------|-----------------------------|
| 1. $\{v_1\}$ | 6. $\{v_2, v_3\}$ |
| 2. $\{v_1, v_2\}$ | 7. $\{v_2, v_3, v_4\}$ |
| 3. $\{v_1, v_2, v_3\}$ | 8. $\{v_2, v_3, v_4, v_1\}$ |
| 4. $\{v_1, v_2, v_3, v_4\}$ | |
| 5. $\{v_2\}$ | |

- | | |
|------------------------------|------------------------------|
| 9. $\{v_3\}$ | 13. $\{v_4\}$ |
| 10. $\{v_3, v_4\}$ | 14. $\{v_4, v_1\}$ |
| 11. $\{v_3, v_4, v_1\}$ | 15. $\{v_4, v_1, v_2\}$ |
| 12. $\{v_3, v_4, v_1, v_2\}$ | 16. $\{v_4, v_1, v_2, v_3\}$ |

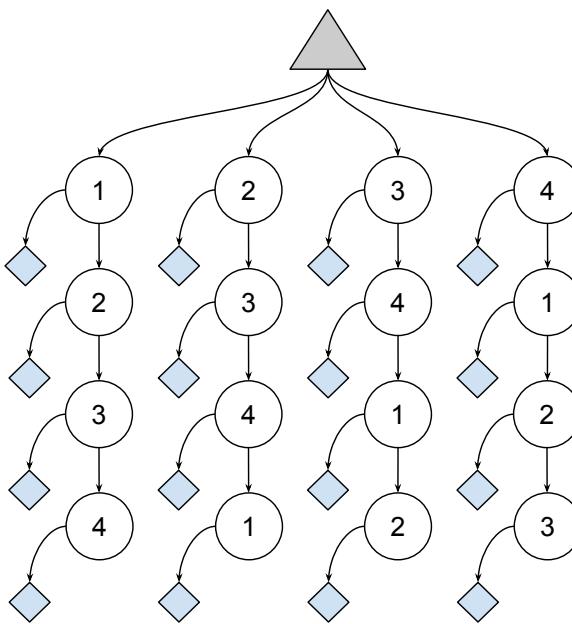


Figura 4. Exemplo do espaço de solução em forma de árvore para o grafo na figura 2. Cada caminho até uma folha é considerado uma possível solução para o problema.

Para verificar se um subgrafo conexo contém todos os focos de mosquito é realizado a inclusão de todos os focos de cada um dos vértices em um conjunto, e então é verificado se esse conjunto é igual ao conjunto total de focos F . O algoritmo 1 exemplifica o processo de análise de cobertura. O processo é bem simples de ser realizado, e tem custo linear.

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 Falso

```

O processo completo é apresentado no algoritmo 2, tendo como entrada o grafo e o conjunto de focos. É interessante ressaltar que a implementação gera o conjunto \mathbb{S}

de forma iterativa, não sendo necessário seu armazenamento na memória. Mais detalhes serão informados na seção sobre implementação.

Algorithm 2: ForcaBruta

Data: $G(V, A), F$
Result: V^*

```

1 begin
2    $\mathbb{S} \leftarrow geraTodosSubgrafosConexos(G)$ 
3    $G^* \leftarrow \emptyset$ 
4   foreach  $G' \in \mathbb{S}$  do
5     if cobreTodosFocos( $G', F$ ) then
6        $G^* \leftarrow melhorSolucao(G^*, G')$ 
7   /*  $V^*$  é o conjunto de vértices de  $G^*$  */
```

return V^*

Casos de empate, ou seja, caso exista dois ou mais elementos de \mathbb{S} que atendem as exigências do problema, são utilizado quesitos de desempates presentes na função *melhorSolução*(G'_a, G'_b). Esses quesitos avaliam o desempate entre dois subgrafos distintos G'_a, G'_b da seguinte forma:

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 vértices 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 vértice 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.

A fim de melhorar os resultados, foram utilizados operações de podas na árvore de soluções, de modo a resuzir o espaço de buscas. Duas abordagens foram utilizadas. A primeira consiste em remover antes de iniciar as buscas todas as folhas de caminhos G' na árvore que tenham tamanho $|G'| \times 2 < |F|$, pois se cada vértice atende somente dois focos, então é necessário no mínimo $\frac{|F|}{2}$ vértices na solução. A figura 5 mostra o resultado após a poda.

A segunda melhoria consiste, ao longo da detecção de soluções viáveis, descartar possíveis caminhos que tenham tamanho maior que a melhor solução encontrada até então. Tal poda impede que ramos sejam avaliados até o final caso já encontraram a solução, e geralmente reduzem consideravelmente o tempo de busca, principalmente para entradas em que $|F|$ é bem menor que $|V|$. A figura 6 exemplifica o espaço de busca após encontrar a primeira solução viável. As folhas em cinza são as ainda não avaliadas, as em vermelho as soluções não viáveis e as em verde as viáveis. A folha em azul é a melhor solução viável encontrada até o momento.

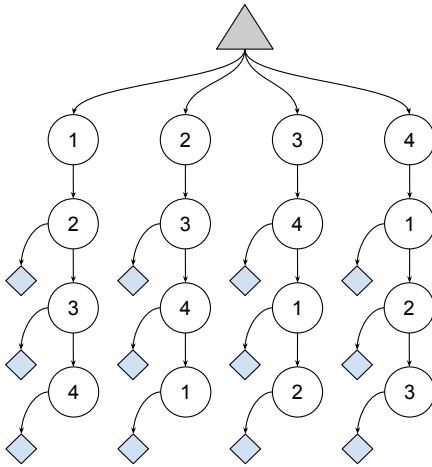


Figura 5. Exemplo do espaço de solução após a poda por mínimo de vértices requerido.

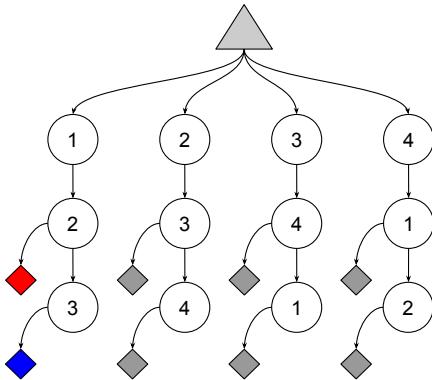


Figura 6. Exemplo do espaço de solução após a poda por limite superior da solução encontrada ao encontrar a primeira solução viável.

2.2.2. Programação Dinâmica

Para resolver o problema proposto através da programação dinâmica é utilizado a idéia de armazenamento de resultados computados anteriormente para reduzir o custo das operações. Assim sendo, o espaço de busca de soluções é explorado de forma mais barata.

Nessa abordagem, começa-se a analisar todos os subgrafos de tamanho 1, e esses subgrafos são acrescidos do próximo vértice adjacente a direita no ciclo até que se encontre uma solução viável. De início temos então $|V|$ subgrafos de tamanho 1.

Considerando que um dos subgrafos obtidos em uma iteração t dado por S_t é resultados de um subgrafo S_{t-1} obtido no tempo $t - 1$ acrescido de um vértice, temos que a quantidade de focos em S_t é maior ou igual a S_{t-1} , e o conjunto de todos os focos de S_t é a união dos focos de S_{t-1} com os do vértice adicionado v_t , ou $\mathcal{R}(S_t) = \mathcal{R}(S_{t-1}) \cup \mathcal{R}(v_t)$. Considerando que já temos $\mathcal{R}(S_{t-1})$ calculado, o custo de adicionar dois focos ao conjunto é constante.

A condição de parada é dada em uma iteração t onde ao menos uma solução

viável é encontrada. Isso ocorre pois qualquer subgrafo obtido após t terá mais vértices que a solução já encontrada. Se utilizarmos uma estrutura de conjuntos que propicie o armazenamento da cardinalidade desse conjunto para verificar a condição de parada ($|\mathcal{R}(S_t)| = |F|$), também podemos obter a cardinalidade em tempo constante, melhorando assim o custo em relação a abordagem por força bruta.

Algorithm 3: Dinâmica

```

Data:  $G(V, A), F$ 
Result:  $V^*$ 

1 begin
2    $\mathbb{S} \leftarrow geraTodosSubgrafosUnitarios(G)$ 
3    $F_{\mathbb{S}} \leftarrow RelacaoFocosSubsets(\mathbb{S})$ 
4    $G^* \leftarrow \emptyset$ 
5   while True do
6     foreach  $G' \in \mathbb{S}$  do
7       if cobreTodosFocos( $G', F, F_{\mathbb{S}}$ ) then
8          $G^* \leftarrow melhorSolucao(G^*, G')$ 
9       if  $G^* \neq \emptyset$  then
10        Break
11     foreach  $G' \in \mathbb{S}$  do
12       AcrescentaProximoVertice( $G', G$ )
13       AtualizaRelacaoFocos( $G', F_{\mathbb{S}}$ )
14   /*  $V^*$  é o conjunto de vértices de  $G^*$  */ 
return  $V^*$ 

```

O algoritmo 3 representa o processo de obtenção de uma solução através do paradigma de programação dinâmica. Nota-se que o algoritmo de avaliação da melhor solução entre as viáveis utiliza o mesmo princípio de desempate adotado pela força bruta. Para o grafo representado na imagem 2 temos os seguintes resultados a cada iteração:

Iteração 1

1. $G_{11} = \{v_1\}$, $\mathcal{R}(G_{11}) = \{f_1, f_2\}$
2. $G_{12} = \{v_2\}$, $\mathcal{R}(G_{12}) = \{f_2, f_3\}$
3. $G_{13} = \{v_3\}$, $\mathcal{R}(G_{13}) = \{f_3, f_4\}$
4. $G_{14} = \{v_4\}$, $\mathcal{R}(G_{14}) = \{f_3, f_4\}$

Iteração 2

1. $G_{21} = \{v_1, v_2\}$, $\mathcal{R}(G_{21}) = \{f_1, f_2, f_3\}$
2. $G_{22} = \{v_2, v_3\}$, $\mathcal{R}(G_{22}) = \{f_2, f_3, f_4\}$
3. $G_{23} = \{v_3, v_4\}$, $\mathcal{R}(G_{23}) = \{f_3, f_4\}$
4. $G_{24} = \{v_4, v_1\}$, $\mathcal{R}(G_{24}) = \{f_1, f_2, f_3, f_4\}$ **Solução**

2.2.3. Algoritmo Guloso

A estratégia gulosa utiliza o princípio de construção de uma solução através da inclusão de elementos através da melhor solução local até que se alcance o objetivo. Para o nosso problema, o objetivo é cobrir todos os focos de mosquito através da quantidade mínima de voluntários (vértices no grafo). O objetivo é determinar as ações que avaliarão qual é o melhor vértice inicial para se começar a estratégia e como deverá ser o critério de inserção de novos vértices.

Para se definir o vértice da solução inicial, realizamos uma busca em largura em cada vértice, e essa busca termina assim que todos os focos forem adicionados a solução da busca e largura. Desse modo, conseguimos encontrar o vértice que está mais próximo em ambas as direções de caminhamento no anel apenas avaliando quantos vértices foram necessários para cobrir todos os focos. Em caso de empate, consideraremos o vértice em que a soma de todos os índices dos elementos adicionados durante a busca em largura seja menor.

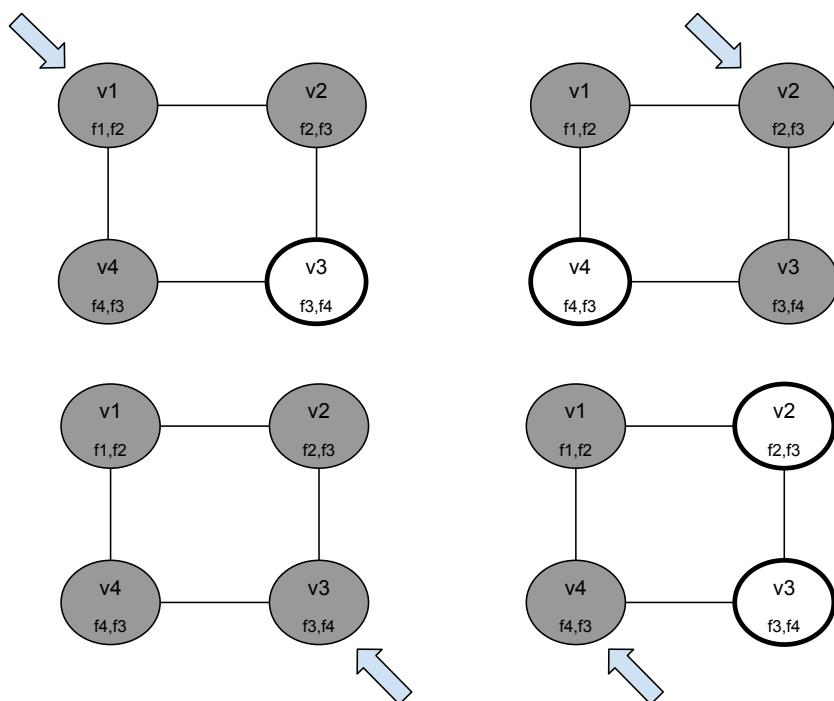


Figura 7. Exemplo de resultado de busca em largura para cada vértice até que se encontre todos os focos.

A figura 7 exemplifica os resultados das buscas em largura para cada um dos vértices de um grafo. Os vértices em cinza são os acrescidos na solução da busca em largura até a condição de parada. O custo de se realizar essa operação de busca é linear em relação a quantidade de vértices para cada busca. Os seguintes resultados de elementos em cada vértice foram encontrados: $v_1 = 3$, $v_2 = 3$, $v_3 = 4$, $v_4 = 2$. Assim sendo, v_4 é a melhor solução iniciada encontrada. Nota-se que a busca em largura avalia primeiro um vértice a direita e então o da esquerda. No caso dado pelo exemplo, quando foi avaliado

após a primeira inserção partindo de v_4 como raiz, não foi necessário verificar o vizinho a esquerda, dado que todos os focos haviam sido encontrados.

Tendo definido o vértice inicial $v_{inicial}$ pertencente ao conjunto solução provisória \mathbb{S}^* , define-se então os dois possíveis candidatos vizinhos da solução a serem incluídos, o adjacente a direita e o adjacente a esquerda do subgrafo \mathbb{S}^* , dados respectivamente por v_d e v_e . Realizamos então duas buscas em largura distintas a partir de v_d e v_e até que todos os focos que ainda não foram cobridos pela solução sejam encontrados, ou seja, a condição de parada é encontrar todos os focos presentes em $F - \mathcal{R}(\mathbb{S})$. O que conter a menor solução será acrescido a solução \mathbb{S} . Em caso de empate será considerado o que estiver mais próximo de $v_{inicial}$, ou o que levar a menor soma de índices em \mathbb{S} caso o empate persista.

Algorithm 4: Guloso

```

Data:  $G(V, A), F$ 
Result:  $V^*$ 
1 begin
2    $melhorCusto \leftarrow \infty$ 
3   foreach  $v' \in G$  do
4      $custo \leftarrow |BFS(v', G, F)|$ 
5     if  $custo < melhorCusto$  then
6        $melhorCusto \leftarrow custo$ 
7        $v_i \leftarrow v'$ 
8    $\mathbb{S} \leftarrow \{v_i\}$ 
9   while  $\mathcal{R}(\mathbb{S}) \neq F$  do
10     $v_d, v_e \leftarrow adjacentes(\mathbb{S})$ 
11     $custo_d \leftarrow |BFS(v_d, G, F - \mathcal{R}(\mathbb{S}))|$ 
12     $custo_e \leftarrow |BFS(v_e, G, F - \mathcal{R}(\mathbb{S}))|$ 
13    if  $custo_d$  melhor que  $custo_e$  then
14       $\mathbb{S} \leftarrow \mathbb{S} \cup v_d$ 
15    else
16       $\mathbb{S} \leftarrow \mathbb{S} \cup v_e$ 
return  $\mathbb{S}$ 
```

O algoritmo 4 representa o pseudo-código da estratégia gulosa utilizada.

3. Análise de Complexidade

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 temporal e espacial dos algoritmos como apresentado nas subseções que seguem.

3.1. Análise de Complexidade Temporal

3.1.1. Força Bruta

O custo de se analizar se uma solução é viável para o algoritmo de força bruta é de custo $O(|V| + |F|)$, dado que é necessário percorrer todos os vértices, inserir seus elementos em um vetor de focos alcançados e depois percorrer essa estrutura verificando se todos os focos dados na entrada são alcançados com esse conjunto.

Como apresentado anteriormente, são contidos num grafo anel $|V|^2$ casos de análise diferentes, logo é necessário que sejam realizadas uma avaliação de custo $O(|V| + |F|)$ para cada uma. Assim sendo, temos que a complexidade total do algoritmo por força bruta para resolver o **ZikaZeroAnelDual** é:

$$O(|V|^3 + |V|^2|F|)$$

3.1.2. Programação Dinâmica

A estratégia por programação dinâmica realiza para cada possível solução a verificação se tal cobre todos os focos. Assim sendo, seu custo é da ordem de $O(|F|)$. Analisar soluções de um certo tamanho consiste em complementar uma solução anterior armazenada com um vértice. Tal operação tem custo constante. Considerando que no pior caso podem ser analisados até $|V|^2$ soluções, poderemos inferir então o custo final da abordagem como:

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

3.1.3. Algoritmo Gulosso

A abordagem gulosa realiza a construção da solução de uma maneira diferente: primeiro ela encontra o melhor vértice para iniciar a solução e incrementa a solução parcial até que obter um conjunto satisfatório para determinar a solução final.

Como durante a obtenção do primeiro vértice é realizado uma busca em largura para cada vértice, e o custo de busca em um grafo anel verificando a cada vértice se todos os focos foram alcançados é $O(|V||F|)$, o custo de obtenção do primeiro vértice é $O(|V|^2|F|)$.

O custo de incremento de um novo vértice a solução é de $2 \times |V||F|$, já que mais duas buscas são realizadas em cada etapa. Logo, no pior caso, podemos ter até $|V| - 1$ vértices incluídos a solução. Assim sendo, a ordem de complexidade de inclusão de todos os outros elementos é $O(|V|^2|F|)$. Por fim, O custo do algoritmo guloso é dado por:

$$O(|V|^2|F|) + O(|V|^2|F|) = O(|V|^2|F|)$$

3.2. Análise de Complexidade Espacial

Para todos os algoritmos foram utilizados uma estrutura de dados que armazena o grafo de entrada em ordem de complexidade $O(|V|)$, considerando que a quantidade de focos

em um vértice é constante.

3.2.1. Força Bruta

Para cada solução avaliada pelo força bruta é necessário uma estrutura para armazenar os vértices e outra para armazenar os focos encontrados. No pior caso elas podem conter $|V|$ vértices e $|F|$ focos. Também é necessário uma estrutura para armazenar a solução melhor já encontrada. Assim sendo, o custo espacial é dado por $O(|V| + |F|)$.

3.2.2. Programação Dinâmica

A programação dinâmica se caracteriza por converter custo computacional em custo de armazenamento, por manter resultados de contas préviamente realizadas. O algoritmo utilizado mantém todas as soluções da etapa anterior, ou seja, $|V|$ soluções. Considerando que cada solução requer espaço $O(|V| + |F|)$, o custo de se armazenar os dados durante a abordagem dinâmica de solução é dada por $O(|V|^2 + |V||F|)$.

3.2.3. Algoritmo Guloso

O algoritmo guloso é dividido em duas etapas: a identificação do melhor vértice inicial e o incremento da solução parcial. A primeira etapa realiza uma busca em largura que requer custo espacial de ordem $O(|V| + |F|)$ para verificar os vértices e armazenar os focos encontrados. A segunda etapa requer o dobro de espaço de uma busca em largura, pois realiza a mesma duas vezes e precisa manter os resultados para fazer a comparação do melhor elemento a se incluir no conjunto solução. Por fim, é necessário armazenar o conjunto solução atual dos focos que o mesmo abrange. Assim, sendo, o custo espacial total para essa abordagem é dado por $O(|V| + |F|)$.

4. 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 *Algoritmos*, que contem a implementação de cada uma das estratégias apresentadas para a solução do problema.

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 *Algoritmos* contém três funções públicas relativas ao algoritmo de Força Bruta, de Estratégia Gulosa e Programação Dinâmica, com interfaces simples de passagem de parâmetros. As funções requerem um apontador para um grafo da classe *Graph* como entrada dos parâmetros, e retorna através da saída padrão (standart output) a lista dos índices dos vértices. A mesma foi implementada em um arquivo denominado *algoritmos.h*.

O programa principal encontra-se no arquivo *main.cpp*, e contém a função main principal responsável por gerenciar as entradas e saídas de dados e o chamado das funções de solução do problema.

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-bruto.sh p*
- *\$ compilar-dinamico.sh*
- *\$ compilar-guloso.sh*

- **Executar:**

- *\$ executar-bruto.sh arquivo_de_entrada arquivo_de_saida*
- *\$ executar-dinamico.sh arquivo_de_entrada arquivo_de_saida*
- *\$ executar-guloso.sh arquivo_de_entrada arquivo_de_saida*

5. Testes de Execução 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.

Os testes de corretude utilizados foram realizados de forma automatizada. Para tal, foram gerados diversas instâncias com variações de quantidade de vértices entre 10 e 5000, e com a quantidade de focos variando entre 2 e $|V|$. Considerando que o algoritmo de força bruta avalia todas as possíveis soluções e escolhe a melhor, definimos seu resultado como o esperado. Assim sendo, todos os resultados obtidos pelas outras abordagens (Força Bruta com podas, Dinâmica e Gulosa) devem levar ao mesmo resultado. Assim sendo, todos os resultados computados pelas intâncias de entrada foram iguais consolidando a corretude dos algoritmos.

A fim de verificar o comportamento de execução do algoritmo acima proposto implementado em uma linguagem de programação, foram realizados os testes de execução. O objetivo principal é verificar se a ordem de complexidade dos algoritmos calculada correspondem aos custos de tempo de execução obtidos. Foram analizados quatro implementações de soluções diferentes para o problema: A força bruta contendo podas, a força bruta sem podas, o algoritmo utilizando programação dinâmica e o algoritmo guloso.

O primeiro teste utilizou instâncias variando a quantidade de vértices e mantendo uma quantidade aleatória de focos entre 2 e a quantidade de vértices no grafo. Assim, podemos ver de forma generalista como o programa se comporta em relação a quantidade de vértices na entrada, independente da quantidade de focos. A figura 8 mostra os resultados obtidos. Podemos notar que a programação dinâmica foi a que obteve os melhores resultados, seguido pelo algoritmo guloso e pelas estratégias utilizando força bruta. Podemos notar então que as estratégias de podas foram muito eficientes para o força bruta, reduzindo seu custo em mais de três vezes (3x).

Para analisar o comportamento do algoritmo quando se encontra uma solução no início da busca, realizamos um experimento que mantém uma quantidade fixa de focos e

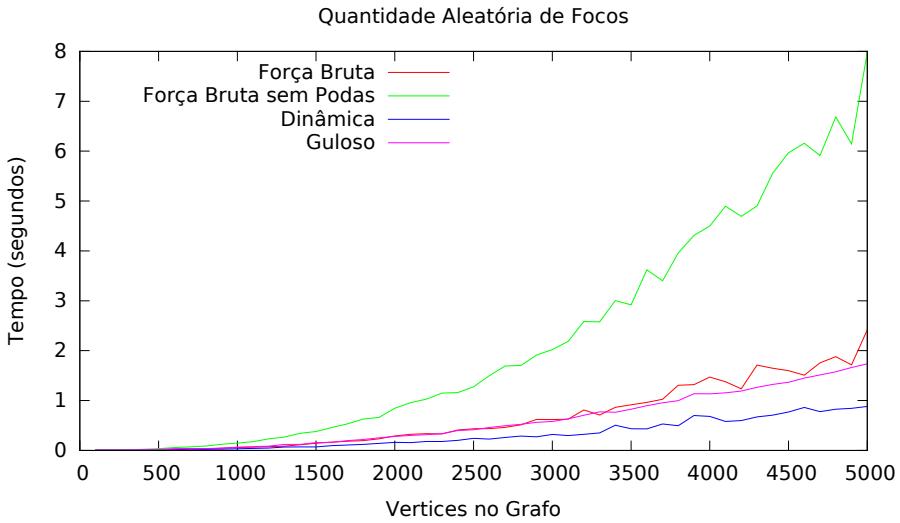


Figura 8. Tempo de execução das propostas apresentadas para casos médios.

aumenta-se gradativamente a quantidade de vértices. Foram utilizados 100 focos em cada grafo de entrada. Analizando a figura 9 podemos notar que a poda é extremamente eficaz para o força bruta, já que o algoritmo converge mais rapidamente para uma solução. O mesmo é notado para os algoritmos guloso e de programação dinâmica. Como a convergência para a solução é bem rápida para esses casos, podemos considerar que as três melhores abordagens obtém tempos similares. O algoritmo por força bruta convencional é extremamente ineficaz nesse caso.

De forma inversa ao teste realizado anteriormente, propomos intâncias com grande quantidade de focos para analizar o comportamento dos algoritmos nesse caso. Consideramos a quantidade de focos igual a quantidade de vértices no grafo, e geramos entradas com grafos de diversos tamanhos. Os resultados podem ser observados na figura 10. É interessante notar que nesse caso o custo das várias abordagens são mais próximos entre si, e a discrepância entre a força bruta e a sua versão com podas é bem menor, já que a segunda estratégia de poda é muito menos utilizada.

Por fim, analizamos o comportamento do algoritmo em um caso específico, a variação da quantidade de focos para uma mesma quantidade de vértices. A imagem 11 mostra os resultados obtidos. Podemos notar que os algoritmos de programação dinâmica e guloso se comportaram uniformemente, enquanto os algoritmos de força bruta tiveram uma variação inesperada, principalmente no meio do gráfico, onde o tempo decai com o aumento de focos. Podemos justificar esse evento por uma característica da estratégia desenvolvida durante a implementação do algoritmo. A estratégia de comparação para verificação se a solução alcançada é ótima consiste em verificar se todos os focos presentes no domínio do grafo estão na solução avaliada, e se caso um deles não estiver a computação é abortada para a avaliação. Como soluções com menos focos convergem para a característica de conter todos os focos mais rápidos, elas requerem uma comparação de todos os focos. Além disso, há muito mais empates, sendo necessário executar a função de desempate mais vezes.

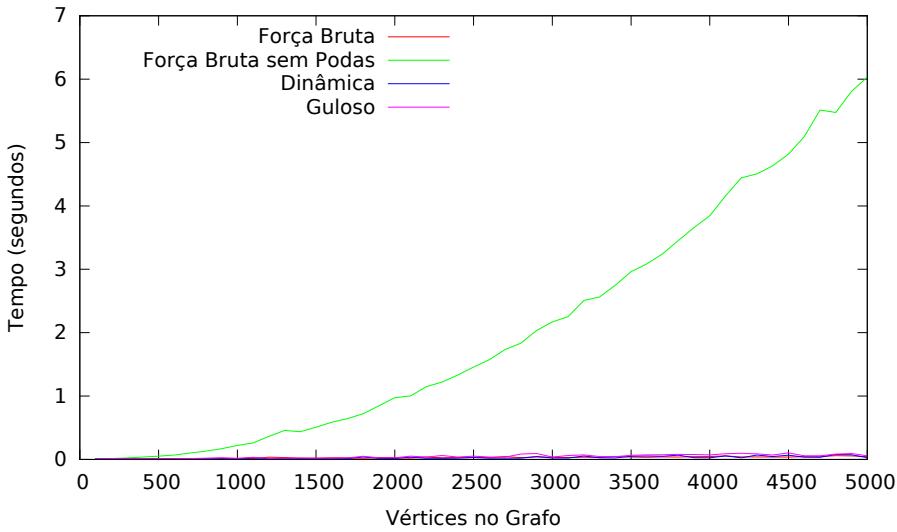


Figura 9. Tempo de execução para quantidade fixa de focos (100 focos).

6. Resultados Obtidos e Conclusões

Após a realização de vários testes, podemos observar que os algoritmos propostos com melhorias se comporta bem e tem seu custo previsível em diversos cenários. A busca pela solução entre todas as possibilidades pode ser extremamente custosa, e sua versão com podas foi extremamente eficiente, com custos comparáveis as outras abordagens.

Podemos notar que os algoritmos são sensíveis a diversas variações de entradas, como o tamanho do grafo e a quantidade de focos a serem cobertos. Percebemos também que entradas com mais vértices são mais custosas que os com menos, e a variação linear da quantidade de focos nem sempre corresponde a um crescimento linear ou uniforme do custo para o algoritmo de força bruta, devido a características da implementação.

Por fim, podemos notar que a implementação utilizando programação dinâmica se mostrou a menos custosa nas diversas situações avaliadas. A estratégia através de um algoritmo guloso também teve um bom resultado, mantendo seu tempo uniforme a variações, embora sempre acima do custo da estratégia dinâmica. Características específicas do problema também facilitaram a obtenção de algoritmos eficazes e eficientes de custo polinomial para a solução do problema.

Referências

- Cormen, T. H. (2002). *Algoritmos: teoria e prática*. Elsevier.
- Gross, J. L. and Yellen, J. (2004). *Handbook of graph theory*. CRC press.
- Shuai, T.-P. and Hu, X.-D. (2006). Connected set cover problem and its applications. In *Algorithmic Aspects in Information and Management*, pages 243–254. Springer.

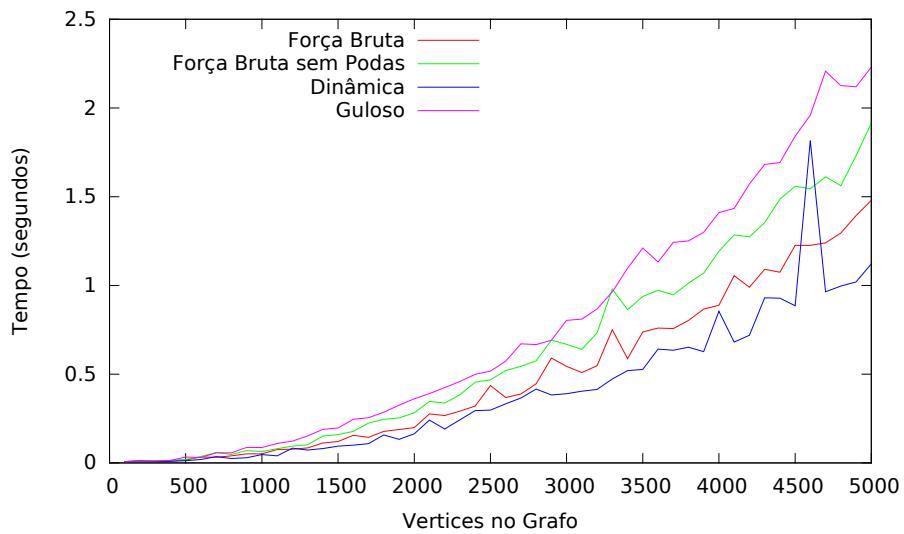


Figura 10. Tempo de execução para a quantidade de focos iguais a quantidade de vértices.

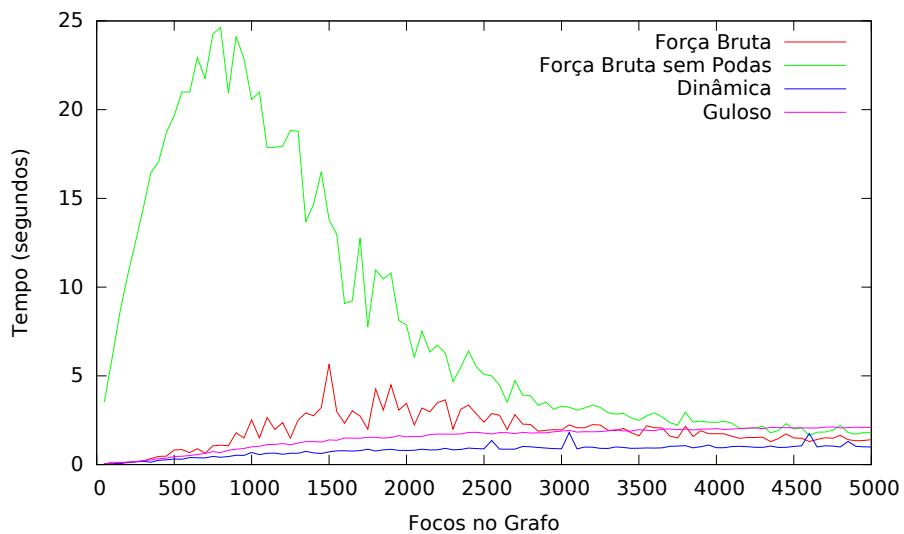


Figura 11. Tempo de execução para entradas variando a quantidade de focos em um grafo de 5000 vértices.

TP Paradigmas - ZikaZeroZ

Alex de Paula Barros¹

¹Universidade Federal de Minas Gerais
Belo Horizonte, MG, Brasil

{alexbarros}@dcc.ufmg.br

Exercício 1

1.1 Tentativa e Erro

O algoritmo de tentativa e erro foi implementado utilizando várias execuções de uma heurística gulosa que explora a topologia anelar do grafo de entrada. Basicamente, a heurística gulosa caminha no grafo sempre em sentido horário até encontrar uma solução viável, que consiste na melhor solução para o problema começando (ou terminando) em um vértice v . Essa heurística é aplicada utilizando todo $v \in \mathcal{V}$ como vértice inicial, obtendo assim a melhor solução iniciada em cada vértice. Como a melhor solução iniciada com cada vértice foi verificada, a melhor destas soluções é também a solução ótima do problema.

Algorithm 1 Exhaustive Search Solution

```
for v in V do
    S = {v}
    next = v.right
    while !isSolution(S) do
        S = S ∪ {next}
        next = next.right
    updateSolution(S)
    S = ∅
```

1.2 Guloso

A solução gulosa consiste em uma heurística que utiliza como critério de seleção o número de novos focos cobertos ao selecionar um novo voluntário. Basicamente, esta heurística seleciona inicialmente o voluntário que tem acesso ao maior número de focos, e em seguida seleciona continuamente o voluntário, vizinho aos já selecionados, que têm acesso ao maior número de focos que ainda não são acessíveis.

1.3 Programação Dinâmica

O algoritmo de programação dinâmica é uma extensão do algoritmo de tentativa e erro, que utiliza o custo da melhor solução encontrada como um limite superior, para o resultado para interromper a heurística gulosa se o custo desta exceder esse limite.

Algorithm 2 Greedy Heuristic

```
startNode = getMostRelevantNode()
left = startNode.left
right = startNode.right
S = {startNode}
while !isSolution(S) do
    if relevance(left) > relevance(right) then
        S = {left}
        left = left.left
    else
        S = {right}
        right = right.right
```

Algorithm 3 Dynamic Programming Solution

```
for v in V do
    S = {v}
    next = v.right
    while !isSolution(S) do
        S = S ∪ {next}
        next = next.right
        |S| > |BestSolution|
    updateSolution(S)
    S = ∅
```

Exercício 2

2.1 Tentativa e Erro

O algoritmo de tentativa e erro utiliza dois laços (for e while) que executam $|\mathcal{V}|$ vezes no pior caso, as operações internas ao while realizam um número constante de operações. A função de atualização da melhor solução é linear em $|\mathcal{V}|$. Assim a complexidade de tempo deste algoritmo é $O(|\mathcal{V}|^2)$.

2.2 Guloso

A seleção de nó inicial é obtida verificando todos os vértices, e portanto é linear no número de vértices, a função de relevância (*relevance()*) tem custo $O(|\mathcal{F}|)$, e é chamada duas vezes a cada iteração do loop, que por sua vez executa $|\mathcal{V}|$ vezes no pior caso. Logo sua complexidade de tempo é $O(|\mathcal{V}| * |\mathcal{F}|)$. Note que a complexidade desta heurística pode ser pior que a do algoritmo de tentativa e erro que obtém a solução ótima. Uma outra opção de heurística de complexidade linear utilizada no problema de tentativa erro.

2.3 Programação Dinâmica

A complexidade do algoritmo de programação dinâmica, no pior caso, é a mesma que a do algoritmo guloso, já no melhor caso, em que o primeiro vértice avaliado é solução, essa complexidade é linear. Assim este algoritmo é $\Omega(|\mathcal{V}|)$ e $O(|\mathcal{V}|^2)$.

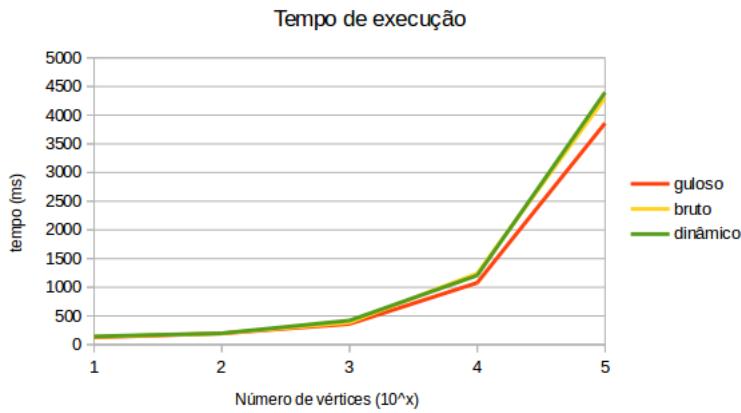


Figura 1. Tempo de execução do algoritmo em relação ao tamanho da entrada

Exercício 5

As soluções propostas foram testadas utilizando instâncias providas com para verificação da corretude dos algoritmos. Para avaliar o tempo de execução dos algoritmos foram geradas 5 grafos aleatórios variando a o número de vértices de cada grafo. A figura 1 apresenta o tempo de execução das três algoritmos. O algoritmo dinâmico e o de força bruta apresentam praticamente o mesmo tempo de execução, o que demonstra que a poda utilizada no algoritmo dinâmico não representou um ganho significativo. O tempo de execução do guloso embora tenha sido o menor que os demais, como a diferença foi muito pequena, não justificou sacrificar a solução ótima com o objetivo de reduzir o tempo de execução.