Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

PROJETO E ANÁLISE DE ALGORITMOS

Segundo Trabalho: disponível em:
http://www.dcc.ufmg.br/~lcrocha/paa/tp2

Código Fonte: disponível em:
http://www.dcc.ufmg.br/~lcrocha/paa/tp2/src

Leonardo Chaves Dutra da Rocha

Professor - Nivio Ziviani

Belo Horizonte
19 de junho de 2006

Resumo

Neste trabalho estudamos dois problemas clássicos da computação: Distância de Palavras e Conjunto Máximo de Vértices Independentes. O primeiro problema consiste em, dadas duas cadeias de caracteres, X e Y, determinar qual o número de operações de retirada, inserção e substituição necessárias para transformar X em Y. Para esse problema implementamos uma solução baseada em programação dinâmica que consiste em resolver incrementalmente problemas menores até se atingir uma solução ótima global, nesse caso a solução utilizada foi a sugerida por Levenshtein. O segundo problema é um problema NP-Completude e consiste em, dado um grafo de V vértices e A arestas, obter o maior conjunto de vértices do grafo tal que para qualquer par de vértices desse conjunto, não existam arestas entre eles. Para resolver esse problema implementamos alguns algoritmos: um que obtém a resposta ótima para o problema mas que têm um custo exponencial; um guloso de custo polinomial as que não possui uma garantia mínima da qualidade da solução; um de custo polinomial e que garante uma qualidade mínima, mesmo que baixa; e por fim um algoritmo genético. Apresentamos uma análise para cada um desses algoritmos, além de uma comparação entre os mesmos.

Sumário

1 Programação Dinâmica

1.1 Introdução

Nessa seção apresentamos um algoritmo que utiliza programação dinâmica para resolver o problema de distância entre strings [35]. Esse problema consiste do seguinte: Sejam dadas duas cadeias de caracteres, X = $x_1x_2x_3x_4...x_m$ e Y=$y_1y_2y_3y_4...y_n$. O número de operações de retirada, inserção e substituição necessárias para transformar X em Y é conhecido como distância de edição, em inglês *edit distance*, ou distância de Levenshtein que advém do cientista russo Vladimir Levenshtein, que considerou esta distância já em 1965. Assim a distância de edição $ed(X, Y)$ corresponde ao número K de operações necessárias para converter X em Y [3, 32].

Vamos tomar como exemplo, se $X = matranda$ e $Y = saturadas$, então $ed(X, Y) = 4$. Nesse caso, a seqüência de operações que devem ser feitas são: (i) substitui x_1 por y_1(s), (ii) insere y_4(u) após x_3, (iii) retira x_6(n) e (iv)insere y_9(s) após x_8.

Além de corretores ortográficos, outra boa aplicação prática desse problema é na área biológica, mais especificamente em análise de cadeias de DNA. Temos que as cadeias de DNA são definidas pelo seguinte alfabeto:

- A = ADENINA
- C = CITOSINA
- G = GUANINA
- T = TIMINA

Considerando uma seqüência origem S1 = GCATAT e uma seqüência destino S2 = GCTAAT, para transformar uma seqüência em outra poderíamos:

- Inserir um A em S2 antes do primeiro T e deletar o A seguinte, uma inserção + uma deleção = custo = 2.
- Deletar o primeiro T e depois inserir um novo T entre as duas letras A, uma deleção + uma inserção = custo = 2.

É possível criar outros modos de transformação para esse exemplo, mas todos resultam em um custo superior a 2, como a *edit distance* é dada pelo custo mínimo esse fica sendo seu valor.

1.2 O Algoritmo

Os algoritmos mais antigos da área são baseados na técnica de programação dinâmica. Essa técnica consiste em resolver incrementalmente problemas menores até se atingir uma solução ótima global para as cadeias que vão sendo separadas. A Figura 1 mostra a matriz de programação dinâmica, estrutura sobre a qual o algoritmo trabalha.

Nossa implementação é baseada na apresentada em [31, 32]. Primeiramente criamos uma matriz de tamanho (M+1)x(N+1), onde M é o tamanho da cadeia de caracteres 1 e N é o tamanho da cadeia de caracteres 2, conforme Figura 1. A matriz é percorrida pelo algoritmo do topo à esquerda, em direção à base à direita. Cada

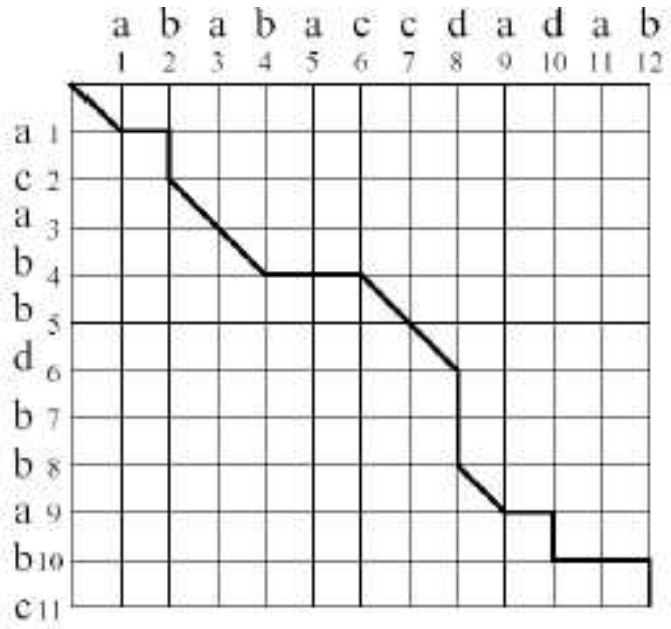

Figura 1: Matriz de programação dinâmica

posição (i, j) da matriz contém a distância de edição entre a cadeia M_0, i e a cadeia N_0, j. Essa distância é calculada da seguinte forma:

1. Compara os caracteres M_i e N_j;
2. Se os dois forem iguais, então custo é igual a 0;
3. Se os dois forem diferentes, então custo é igual a 1;
4. Verifica-se então qual das seguintes células vizinhas da célula em questão é menor: $((M_(i-1), N_j) + 1,\ (M_j, N_(j-1)) + 1,\ (M_(i-1), N_(j-1)) + custo)$
5. Assim o custo da célula em questão é o menor deles.

Na Figura 1 podemos ver que o caminho demarcado representa o conjunto mínimo de operações que transforma a cadeia que rotula as linhas na cadeia que rotula as colunas. O caminho de menor custo entre o topo à esquerda e a base à direita da matriz representa o conjunto de operações que transformam M em N. Uma linha vertical representa a inserção de um caractere em N e uma linha horizontal representa a inserção a remoção de um caractere em N. As substituições ocorrem nas linhas diagonais que percorrem caracteres diferentes.

Em nossa implementação, a matriz de programação dinâmica, ao invés de ser uma matriz do tipo inteiro onde apenas o custo é armazenado, criamos uma matriz do tipo *celula*, que além do custo de operação, guarda também a linha e a coluna da célula que a originou e qual operação deveria ser realizada naquele instante. Dessa forma, não precisamos nos preocupar com a forma com que caminhávamos pela matriz para buscar a solução, basta seguir a linha e a coluna da célula que o originou. Veja por exemplo na Figura 2, como ficaria a nossa matriz caso estivéssemos comparando as palavras (MAT e SUAT):

		S	U	A	T
	0	1	2	3	4
M	1	C_{00} S 1	C_{11} I 2	C_{12} I 3	C_{12} I 4
A	2	C_{11} D 2	C_{11} S 2	C_{12} N 2	C_{23} I 3
T	3	C_{21} D 3	C_{22} D 3	C_{23} D 3	C_{23} N 2

Figura 2: Exemplo Funcionamento da Matriz

Assim, ao final da montagem, basta percorrer a matriz da base a direita em direção a topo esquerda, seguindo as orientações contidas na própria matriz! Um pseudo código para o algoritmo solução descrito acima pode ser visto abaixo:

```
int dista(char string1[1..tamString1],char string2[1..tamString2])
begin
   // c é a matriz com tamString1 +1 linhas e tamString2 +1 colunas
   declare int c[0..tamString1, 0..tamString2]
   // i e j serão utilizados para as interações
   // entre string1 e string2
   declare int i, j, cost

   for i=0 to tamString1
       c[i, 0] = i;
   for j=0 to tamString2
       c[0, j] = j
   for i=1 to tamString1
   begin
       for j=1 to tamString2
       begin
           if string1[i] = string2[j] then
                custo = 0
           else
                custo = 1
           c[i, j].custo = minimo(c[i-1, j  ] + 1,    //delecao
                                  c[i  , j-1] + 1,    //insercao
                                  c[i-1, j-1] + custo //substituicao
                                 )
          if minimo(c[i-1, j  ] + 1,
                    c[i  , j-1] + 1,
                    c[i-1, j-1] + cost) = c[i-1, j  ] + 1 then
                   c[i, j].operacao = delecao
                   c[i, j].linha = i-1
                   c[i, j].cluna = j
```

```
          if minimo(c[i-1, j  ] + 1,
                    c[i  , j-1] + 1,
                    c[i-1, j-1] + cost) = c[i, j-1] + 1 then
                   c[i, j].operacao = insercao
                   c[i, j].linha = i
                   c[i, j].cluna = j-1
          if minimo(c[i-1, j  ] + 1,
                    c[i  , j-1] + 1,
                    c[i-1, j-1] + cost) = c[i-1, j-1] + custo) then
                    if (custo!=0){
                        c[i, j].operacao = substituicao
                    else{
                        c[i, j].operacao = nada
                    }
                   c[i, j].linha = i-1
                   c[i, j].cluna = j-1
       end
    end
   imprime(matriz,tamString1,tamString2);
   return c[tamString1, tamString2].custo
end
```

1.3 Prova de Correção

A seguir apresentamos a prova de correção da transformação do segmento inicial M(1...i) em N(1...j) usando o mínimo de operações c(i,j).custo. Essa prova segue a apresentada em [31, 32].

1. É inicialmente verdadeira na linha e colunas 0 porque M[1..i] pode ser transformado num string vazio N[1..0] simplesmente apagando todos os i caracteres. Do mesmo modo, podemos transformar N[1..0] em M[1..j] simplesmente adicionando todos os caracteres j.

2. O mínimo é tomado em três distâncias, sendo em qualquer das quais possível que:

   (a) Se podemos transformar M[1..i] em N[1..j-1] em k operações, então nós podemos simplesmente adicionar N[j] depois para obter N[1..j] em k+1 operações.

   (b) Se podemos transformar M[1..i-1] em N[1..j] em k operações, então nós podemos fazer as mesmas operações em M[1..i] e depois remover o M[i] original ao fim de k+1 operações.

   (c) Se podemos transformar M[1..i-1] em N[1..j-1] em k operações, então podemos fazer o mesmo com M[1..i] e depois fazer uma substituição de N[j] pelas M[i] originais no final, se necessário, requerendo k+custo operações.

3. As operações requeridas para transformar M[1..n] em N[1..m] é o número necessário para transformar todos os M em todos os N, e logo C[n,m] contém o nosso resultado desejado.

Esta prova não confirma que o número colocado em c[i,j] seja de fato o mínimo; isso é mais difícil de provar e exige um argumento *Reductio ad absurdum* no qual assumimos que C[i,j] é menor do que o mínimo dos três, e usamos isto para mostrar que um dos três não é mínimo. Em [22] existe uma prova detalhada dessa solução, a qual apresentaremos a seguir de forma bastante sucinta.

Lema 1. *O valor de* $C(i,j)$ *deve ser* $C(i-1,j)+1$, $C(i,j-1)+1$ *ou* $C(i-1,j-1)+custo(i,j)$ *e não há outra possibilidade.*

Demonstração. Considere um corretor de edição para a transformação de $M[1..i]$ a $N[1..j]$ por meio do menor número de operações de edição e mantenha atenção no último símbolo. Esse último símbolo pode ser I (inserção), D (deleção), S (substituição) ou N (quando ocorre um casamento, nada a fazer). Se esse último símbolo for um I, então a última operação é a inserção do caractere $N(j)$ no final da primeira string de transformação M. Assim, o corretor antes de I deve especificar o número mínimo de operações para transformar $M[1..i]$ em $N[1..j-1]$. Se não acontecer essa especificação, então a transformação de $M[1..i]$ em $N[1..j]$ será maior que o número de operações mínimo. Por definição, até a penúltima transformação foram necessárias $C(i,j-1)$ operações de edição. Isso faz com que, se o último símbolo do corretor for um I, então $C(i,j)=C(i,j-1)+1$. Da mesma forma, se o último símbolo do corretor foi um D, significa que a última operação foi uma remoção de $M(i)$ e os símbolos do corretor que estão à esquerda de D devem especificar o número de operações mínimo para que possamos transformar $M[1..i-1]$ em $N[1..j]$. Assim, por definição, até a penúltima transformação foram necessárias $C(i-1,j)$ operações de edição, ou seja, se o último símbolo do coretor for D, então $C(i,j)=C(i-1,j)+1$. Mas se o último símbolo do corretor for um S, temos então que a última operação de edição deve substituir $M(i)$ por $N(j)$, e os símbolos à esquerda de S especificam assim, o número mínimo de operações de edição que são necessárias para $M[1..i-1]$ se transforme em $N[1..j-1]$, ou seja, $C(i,j)=C(i-1,j-1)+1$. E por fim, se o último símbolo do corretor for um N, temos que $M(i)=N(j)$ e $C(i,j)=C(i-1,j)$. Utilizando a variável de custo $custo(i,j)$, a qual é 0 se $M(i)=N(j)$ e 1 se $M(i)=N(j)$. Assim temos que se o último símbolo do corretor for um N ou um S, então $C(i,j)=C(i-1,j-1)+custo(i,j)$. Dessa forma cobrimos todos os casos do lema. □

Lema 2. *Temos que* $C(i,j) \leq min[C(i-1,j)+1, C(i,j-1)+1, C(i-1,j-1)+custo(i,j)]$.

Demonstração. Primeiramente temos que $M[1..i]$ pode ser transformada em $N[1..j]$ com $C(i,j-1)+1$ operações. No entanto, é necessário apenas transformar $M[1..i]$ em $N[1..j-1]$ com o número mínimo de operações para que possamos utilizar mais uma operação para inserir o caractere $N(j)$ no final. O número de operações que faz essa transformação de M em N é exatamente $C(i-1,j)+1$ operações. É possível também transformar $M[1..i-1]$ em $N[1..j]$ com o menor número de operações para em seguida removermos o caractere $M(i)$. Assim o número de operações desta

transformação é igual a $C(i-1,j)+1$. Por último podemos fazer a transformação com $C(i-1,j-1)+custo(i,j)$ operações, utilizando o mesmo argumento descrito acima. □

Assim, a partir desses dois lemas, é possível prova o seguinte teorema:

Teorema 1. *Se i e j são ambos positivos, então* $C(i,j) = min[C(i-1,j)+1, C(i,j-1)+1, C(i-1,j-1)+custo(i,j)]$

Demonstração. O Lema 1 diz que $C(i,j)$ precisa ser igual a um dos valores $C(i-1,j)+1, C(i,j-1)+1, C(i-1,j-1)+custo(i,j)$. Já o lema 2 diz que $C(i,j)$ deve ser menor ou igual ao menor a um desses valores. Assim, $C(i,j)$ precisa ser igual ao menor valor dentre os três valores possíveis. □

Ressaltamos que essa é uma descrição sucinta da prova e que maiores detalhes para a mesma pode ser encontrada em [22].

1.4 Análise de Complexidade

Observando o pseudo-código apresentando acima, temos que o parte principal do algoritmo em questão é composto de dois loops aninhados nos quais acontece a montagem da matriz dinâmica que contém a solução. Para cada caractere da primeira cadeia, comparamos com todos os caracteres da segunda cadeia. Essa é a parte essencial do algoritmo, assim temos que o algoritmo direto a partir da técnica de programação dinâmica implementada nessa questão tem uma complexidade de tempo de $O(|n||m|)$, ou seja, $O(n^2)$. Esse algoritmo, além de poder ser usado para o cálculo da distância de edição também pode ser utilizado para descobrir o conjunto de operações que transforma uma cadeia em outra. Adaptações triviais ao algoritmo também permitem sua utilização para a busca aproximada de padrões.

Para verificamos a análise de complexidade de tempo do algoritmo, realizamos alguns experimentos variando o tamanho das strings entre 10 e 8000 caracteres. Na Tabela 1 podemos ver os tempos medidos para cada uma dessas entradas.

Foi feita uma aproximação desses dados com a curva $f(x) = b(x^{(2-a)})$ através de uma regressão utilizando a ferramenta Gnuplot [17]. Dessa forma encontramos a função $f(x) = 8.15599(e-07)(x^{(1.777)})$, ou seja, comprovamos a complexidade de tempo $O(n^2)$. O gráfico dos dados medidos e da curva encontrada pode ser observado na Figura 3, mostrando a convergência das mesmas.

No algoritmo implementado, a maior quantidade da dados armazenados é uma matriz de tamanho (M + 1) X (N +1) a qual armazena a solução do problema. Dessa forma temos que a complexidade de espaço para nosso algoritmo também é $(O(n^2))$.

1.5 Exemplo de Funcionamento

Para mostrar o funcionamento do programa implementado, apresentamos a seguir alguns dos testes realizados, com diferentes palavras.

- Resultado da comparação das palavras matranda e saturadas, conforme solicitado na especificação do trabalho:

Tabela 1: Relação Tempo de Execução pelo Tamanho das Strings

Tamanho da String	Tempo Execução(s)
10	0.000999
20	0.000999
30	0.000999
40	0.000999
50	0.001999
60	0.001999
70	0.001999
80	0.002999
90	0.003999
100	0.003999
200	0.005999
300	0.009998
400	0.017997
500	0.029995
600	0.047992
700	0.069989
800	0.097985
900	0.133979
1000	0.178972
2000	0.356945
4000	1.092833
8000	3.964397

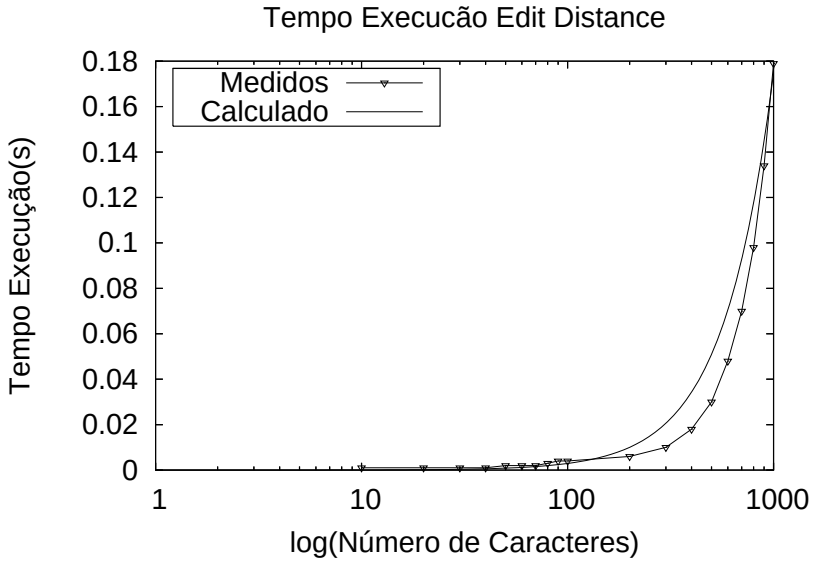

Figura 3: Análise de Complexidade

```
String1: (matranda) String2: (saturadas)
```

```
Serão necessárias 4 operações p/ transforma String1 em String2.
São elas:
Insira String2[9](s) após String1[8](a)
Retire String1[6](n)
Insira String2[4](u) após String1[3](t)
Substitua String1[1](m) por String2[1](s)
```

- Resultado da comparação das palavras bobagem e babonoagem:

```
String1: (bobagem) String2: (babonoagem)
Serão necessárias 4 operações p/ transforma String1 em String2.
São elas:
Insira String2[6](o) após String1[3](b)
Insira String2[5](n) após String1[3](b)
Insira String2[4](o) após String1[3](b)
Substitua String1[2](o) por String2[2](a)
```

- Resultado da comparação das palavras abs e acbd:

```
String1: (abs) String2: (acbd)
Serão necessárias 2 operações p/ transforma String1 em String2.
São elas:
Substitua String1[3](s) por String2[4](d)
Insira String2[2](c) após String1[1](a)
```

- Resultado da comparação das leo abs e Leo, onde ele identifica a diferença entre l maiúsculo e minúsculo:

```
String1: (leo) String2: (Leo)
Serão necessárias 1 operações p/ transforma String1 em String2.
São elas:
Substitua String1[1](l) por String2[1](L)
```

- Resultado da comparação das palavras agata e arragata:

```
String1: (agata) String2: (arragata) tam1:(5) tam2:(8)
Serão necessárias 3 operações p/ transforma String1 em String2.
São elas:
Insira String2[4](a) após String1[1](a)
Insira String2[3](r) após String1[1](a)
Insira String2[2](r) após String1[1](a)
```

1.6 Considerações Finais

Os algoritmos baseados em programação dinâmica não são competitivos para a busca aproximada em relação aos algoritmos mais recentes que combinam paralelismo de bits com filtragem. Porém, é importante levá-los em consideração por diversos

aspectos. O principal deles é a flexibilidade em relação ao custo das operações. Esses algoritmos permitem facilmente modificar o custo de cada operação, e ainda mais, definir custos diferentes para cada operação em função da posição de ocorrência da operação. Outra característica dos algoritmos baseados em programação dinâmica é a facilidade de se recuperar o conjunto de operações que foi utilizado, em geral com uma penalidade de espaço. Algoritmos de outras classes dificilmente podem ser adaptados para obter essa saída adicional.

2 Conjunto de Vértices Independentes

2.1 Introdução

Em teoria de grafos, um conjunto independente de um grafo G(V,A) é um conjunto S de vértices de G tal que não existem dois vértices adjacentes contidos em S [14]. Em outras palavras, se a e b são vértices quaisquer de um conjunto independente, então não há aresta entre a e b. Todo grafo tem ao menos um conjunto independente: o conjunto unitário. Um grafo pode ter vários conjuntos independentes distintos. Se S é um conjunto independente de G e não existe um conjunto independente de G maior que S, diz-se que S é um conjunto independente máximo de G. O problema de, dado um grafo G, determinar um conjunto independente máximo de G é um problema **NP-completo**.

Assim temos a definição: V' é um conjunto independente de vértices de G se somente se $\forall u, v \in V' : u \neq v \Rightarrow (u, v) \notin A$. A partir dessa definirão temos algumas características importantes:

- V' é conjunto independente de G;
- V\V' é a cobertura do vértice G;
- $\forall V'' \subset V' : V''$ é conjunto independente de G;

Este problema possui várias aplicações práticas:

- O conjunto independente de vértices pode ser usado para modelar problemas de escalonamento de recursos. No caso de montagem de um quadro de horários, poderíamos ter como vértices as aulas a serem ofertadas. Para cada professor habilitado para duas ou mais disciplinas (aulas portanto). Teríamos uma aresta ligando as diferentes aulas para as quais ele está habilitado. O conjunto independente de vértices deste grafo mostraria quais aulas devem ofertadas em horários diferentes.
- Suponhamos que você queira realizar uma reunião envolvendo o maior número possível de pessoas do seu círculo de amizades que não se conhecem. Dentre as pessoas que poderiam ser convidadas, você traça um grafo contendo uma aresta ligando duas pessoas que se conhecem. O conjunto independente máximo representa o maior conjunto de pessoas que não se conhecem.
- Outra aplicação para esse problema seria o de identificar localizações para um novo serviço baseado em franquias. Nem um par de franquias pode ser perto o

suficiente para que acabem competindo uma com a outra. Modelando este problema como um grafo onde os vértices são possíveis localizações das franquias e as arestas indicam que estas localizações são demasiadamente próximas, o conjunto independente máximo de vértices irá fornecer o número máximo de franquias que poderão ser abertas sem que haja competição "canibalesca" entre qualquer par de lojas.

- Seja um grafo cujos vértices representam projetos que podem ser executados em uma unidade de tempo. Todo o projeto que utiliza recursos comuns a um outro projeto são interligados por uma aresta. O conjunto independente máximo representa o conjunto maximal de projetos que podem ser executados em paralelo (simultaneamente) em um único período de tempo

- Problema das oito rainhas. Oito rainhas são colocadas em um tabuleiro de xadrez de tal forma que nenhuma rainha possa atacar diretamente outra rainha. Este problema foi investigado por C. F. Gauss em 1850, que não conseguiu resolvê-lo inteiramente. O problema pode ser generalizada para um tabuleiro qualquer de tamanho n x m. Seja um grafo cujos vértices representam as posições de um tabuleiro qualquer. Para cada posição do tabuleiro, interligar por uma aresta todas as posições que possam ser atingidas pela rainha a partir dela. O conjunto independente máximo representa a solução para o problema das n rainhas.

Vejamos um exemplo de conjunto de vértices independentes. Na Figura 4 apresentamos um grafo que contém 6 vértices(a,b,c,d,e,f). Temos que, se tomarmos os vértices (b,d) veremos que não existe nenhuma aresta entre quaisquer pares de vértices desse conjunto. Dessa forma temos que (b,d) é um conjunto independente de vértices. Se tomarmos (a,c,e), também não existe nenhuma aresta entre quaisquer pares de vértices desse conjunto, além disso temos que, para qualquer conjunto de vértices independentes do grafo em questão, nenhum deles é maior que (a,c,e), portanto, temos que esse conjunto também o conjunto de vértices independentes máximo.

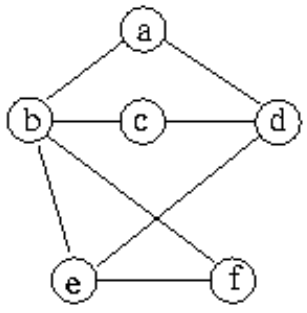

Figura 4: Grafo de Exemplo 1

Tomemos outro exemplo apresentado na Figura 5. O conjunto de vértices independentes do grafo é apresentado na própria figura pelos círculos, e esse conjunto também é o máximo.

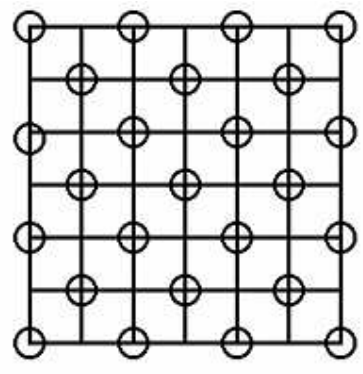

Figura 5: Grafo de Exemplo 2

E por fim apresentamos o grafo de exemplo da questão na Figura 6. Nesse grafo temos que o conjunto de vértices independentes máximo é dado pelos vértices (2,3,4).

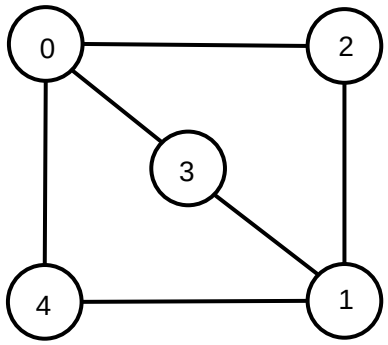

Figura 6: Grafo de Exemplo 3

2.2 Clique de um Grafo

Uma clique em um grafo não-dirigido é um conjunto de vértices dois a dois adjacentes. Em outras palavras, um conjunto C de vértices é uma clique se tiver a seguinte propriedade: para todo par v,w de vértices distintos em C, existe uma aresta com pontas v e w, ou seja é um sub-grafo de completo. O tamanho de uma clique é igual a cardinalidade de seu conjunto de vértice. Exemplo: para qualquer vértice v, o conjunto v é uma clique. Outro exemplo: se o grafo tem alguma aresta com pontas v e w então o conjunto v,w é uma clique [48, 15, 27].

Uma clique C é maximal se não existe clique C' que seja superconjunto próprio de C. Uma clique C é máxima se não existe clique C' que seja maior que C. Encontrar uma clique maximal é fácil (veja próxima seção); encontrar uma clique máxima é mais difícil.

Um exemplo de aplicação de clique é o seu uso na detecção de fraudes em esquemas de reembolso por mal atendimento de um determinado serviço. As requisições são encaminhadas neste sentido são representadas por vértices em um grafo. As arestas são inseridas representando similaridades suspeitas entre as requisições. A detecção de um clique grande no grafo aponta fortemente para uma fraude

Em [40], a construção de cliques é usada no modelamento de proteínas para a análise comparativa de sua estrutura. Neste modelamento, cada possível conformação de resíduo numa seqüência de amino-ácido é representada por um vértice num grafo. Arestas são traçadas entre vértices que sejam mutuamente consistentes. As arestas recebem pesos conforme a interação atômicas entre os átomos representados pelos vértices. Cliques com melhores pesos representam combinações ótimas.

Vejamos um exemplo de clique máximo de um grafo. Na Figura 7 apresentamos um grafo. Para encontramos o clique máximo, devemos encontrar o maior conjunto de vértices tal que para quaisquer par de vértices desse conjunto, existe uma aresta entre eles. Assim para o grafo em questão, temos que o clique máximo é apresentado pela Figura 8.

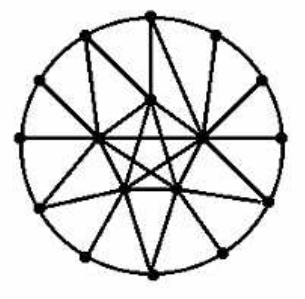

Figura 7: Grafo de Exemplo 4

Figura 8: Clique do Grafo de Exemplo 4

2.2.1 Prova de NP-Completude

Segue abaixo um esboço da prova de que o problema do clique é NP-Completo. A partir da prova desse problema e de uma redução do problema do clique ao problema do conjunto de vértices independentes temos também a prova que o conjunto de vértices independentes é NP-Completo também.

Dado um grafo não orientado G = (V,E) e um valor inteiro $k \in 1, ..., n$, onde $n = |V|$, pergunta-se: G possui uma clique com k vértices:

- Teorema: CLIQUE $\in$ NP-completo.

  1. CLIQUE está NP.

2. SAT $\propto_{poli}$ CLIQUE
   - **Definição**: um grafo G = (V,E) é t-partido se o conjunto de vértices pode ser particionado em t subconjuntos V1,V2, ... ,Vt tal que não existam arestas em E ligando dois vértices em um mesmo subconjunto Vi, i $\in (1, ..., t)$.
   - Transformação de uma instância SAT em uma instância clique: Seja F = C1,C2,...,Cc uma fórmula booleana nas variáveis x1,...,xv. Construa o grafo c-partido G = ((V1, V2, ... ,Vc);E) tal que:
     * Em um subconjunto Vi existe um vértice associado a cada variável que aparece na cláusula Ci de F;
     * A aresta (a,b) estão em E se e somente se a e b estão em subconjuntos distintos e, além disso, a e b não representam simultaneamente uma variável e a sua negação.
3. O número de vértices de G é O(c.v) enquanto o número de arestas é $O(c^2v^2)$. Fazendo-se k = c, teremos construído uma instância de clique em tempo polinomial no tamanho da entrada de SAT.
4. È fácil mostrar que a fórmula F é satisfeita por alguma atribuição de variáveis se e somente se o grafo c-partido G tem uma clique de tamanho c.
5. Exemplo da redução: seja:

$$F = (x_1 + x_2 + \overline{x}_3).(\overline{x}_1 + \overline{x}_2 + x_3).(x_1 + \overline{x}_3)$$

   o grafo corresponde à instância de clique é dada pela Figura 9:

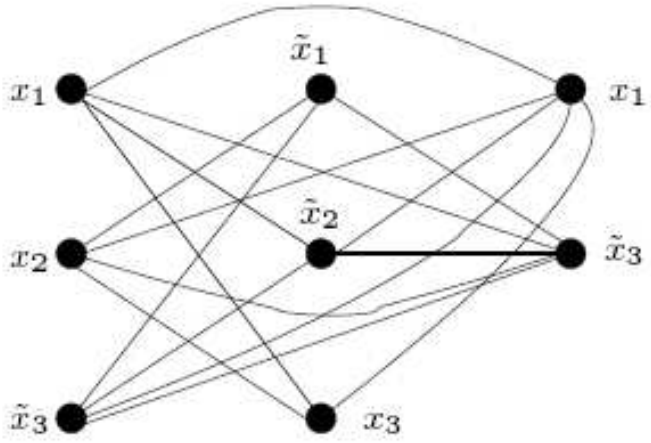

Figura 9: Exemplo de Instância de Clique

O problema do conjunto de vértices independentes máximo de um grafo pode ser polinomialmente reduzido ao problema do clique e vice-versa. Em ambos os casos, a redução polinomial é baseada na inversão de aresta, ou seja, onde existe aresta deixa de existir e onde não existia aresta passa a existir.

Na seção a seguir apresentamos um algoritmo que nos fornece a solução ótima para problema do conjunto de vértices independentes de um grafo. Além disso, através da transformação polinomial descrita anteriormente, aplicamos o mesmo algoritmo para o problema do clique máximo de um grafos

2.3 Aproximação para Conjunto Vértices Independentes

Algoritmos aproximados encontram uma solução com garantia de qualidade em tempo polinomial. Segue abaixo uma prova de que não existe uma aproximação absoluta para o problema do CLIQUE com complexidade polinomial a menos que P = NP. A mesma prova pode ser aplicada ao problema do Conjunto de Vértices Independentes. Essa prova é baseada na prova apresentada por Cid C. de Souza do IC-UNICAMP. Primeiramente, vamos tomar a seguinte nomenclatura descrita na Tabela 2

Tabela 2: Nomenclatura

Nome	Descrição
P	Problema NP-difícil
H	Algoritmo Aproximado
I	Instância do Problema
$z^*(I)$	Valor Ótimo da Instância I
$z^H(I)$	Valor da Solução obtida por H para a Instância I

Dessa forma, temos que a Aproximação Absoluta é dada pela seguinte equação:

$$|z^*(I) - z^H(I)| \leq k \qquad para \;\; algum \;\; k \in \mathbb{Z}_+ \;\; e \;\; para \;\; todo \;\; I$$

Teorema 2. *Não existe uma aproximação absoluta para CLIQUE com complexidade polinomial a menos que $P = NP$.*

Demonstração. Suponha que $P \neq NP$ e que existe um algoritmo polinomial H para CLIQUE tal que $|z^*(I) - z^H(I)| \leq k \in \mathbb{Z}_+$. Seja G^{k+1} o grafo composto de $k+1$ cópias de G mais todas as arestas ligando pares de vértices em diferentes cópias.

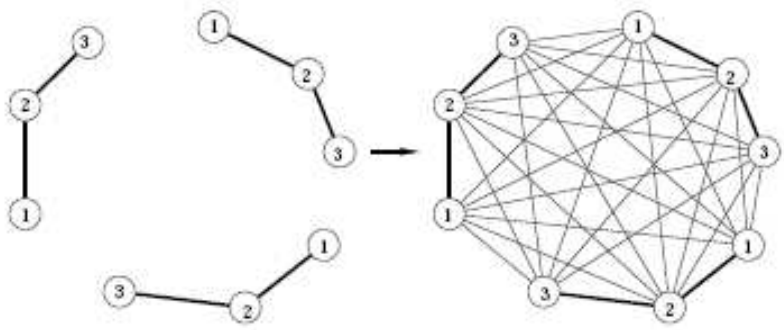

Figura 10: Exemplo

Observação: se α é o tamanho da maior clique de G, então a maior clique de G^{k+1} tem $\alpha(k+1)$ vértices.

Executando-se H para G^{k+1} tem-se que:

$$z^*(G^{k+1}) - z^H(G^{k+1}) \leq k \Longrightarrow z^H(G^{k+1}) \geq (k+1)z^*(G) - k$$

Se C é a clique encontrada por H em G^{k+1}, existe uma cópia de G tal que $C' = V \cap C \;\; e \;\; |C'| \geq |C|/(k+1)$. Logo:

$$|C'| \geq \frac{(k+1)z^*(G) - k}{(k+1)} = z^*(G) - \frac{k}{k+1}$$

Portanto, $|C'| \geq z^*(G)$, ou seja, C' é uma clique máxima de G. No entanto isso é absurdo, logo a suposição inicial é falsa. Logo, não existe uma aproximação absoluta para CLIQUE com complexidade polinomial a menos que P = NP. Por esse mesmo método é possível prova que não existe essa aproximação para o problema do conjunto de vértices mínimos. □

Além da prova apresentada acima, podemos citar Feige [18] e Safra [5] que apresentam uma prova de que o problema do conjunto máximo de vértices independentes não pode se aproximado por um fator de $2^{\frac{\log\log n}{\log\log\log n}}$, a menos que P=NP, além do trabalho de Berman [7] que argumentam que esse problema não pode ser aproximado com nada a menos que seja limitado por um n. No entanto, temos que no trabalho apresentado por Ravi Boppana [8] mostra que para algumas classes de grafo, a aproximação do algoritmo apresentado é dada por $O(\frac{n}{(\log n)^2})$, como veremos mais adiante.

2.4 Solução Ótima

O algoritmo que implementamos que é capaz de obter a solução ótima para o problema de vértices independentes é baseado na técnica de enumeração, ou seja, a partir da enumeração de todas as possibilidades que existem, verifica-se para cada uma delas se ela é possível e se a mesma é maior que maior solução encontrada até aquele momento. Essa técnica também é vulgarmente conhecida como Força Bruta, pois avalia toda as possibilidades existentes.

No arquivo de entrada, temos na primeira linha o número de vértices do grafo e nas linhas posteriores temos um par de números (vértices) que representavam um aresta entre eles. Como no exemplo abaixo que representa o grafo da Figura 6:

```
5
0 2
0 3
0 4
1 2
1 3
1 4
```

Primeiramente, é feita a leitura do arquivo e uma matriz de adjacência de tamanho (número de vértices) X (número de vértices) é criada. Nessa matriz armazenamos as arestas de cada vértice.

Como os vértices são representados por números entre 0 e o (número de vértices - 1), utilizamos um ordenador lexicográfico [46, 41], o qual gera todas as possibilidades existentes de combinação entre esses vértices. Para cada uma dessas possibilidades, verificamos se a mesma é uma solução válida para o problema de vértices independentes ou não.

Essa verificação é feita através de uma função que, para cada par de vértices contido na combinação em questão, ou seja, na solução candidata, se existe uma aresta que os interconecta. Caso exista alguma aresta, abortamos a verificação e a solução candidata deixa de ser candidata e não é mais considerada. Caso a função de verificação não encontre nenhuma aresta que conecta nenhuma par de vértices em questão, temos que o conjunto é uma solução possível. Como o objetivo do programa é encontrar o maior conjunto de vértices independentes, verifica-se então se o tamanho da solução encontrada é maior que a maior solução encontrada até então. Caso a resposta seja afirmativa, a maior solução passa a ser a solução que acabamos de encontrar e em caso negativo, descartamos a solução que acabamos de encontrar. Isso é feito para todas as possíveis combinações e no final da execução temos a solução ótima para o problema. Um pseudo-código para o problema é apresentado a seguir:

```
void forca_bruta()
begin
    tamanho_solucao_final = -1;
    le_arquivo(matriz,Nvertices);
    gera_todas_combinacoes(combinacaoes,Nvertices);
    retorno = obtem_combinacao(combinacoes,solucao_possivel);
    while (retorno>=0)
    begin
      valida = testaPossibilidade(solucao_possivel,tamanho_solucao);
      if (valida){
          if (tamanho_solucao>=tamanho_solucao_final){
              solucao_final = solucao_possivel;
              tamanho_solucao_final = tamanho_solucao;
          }
      }
      retorno = obtem_combinacao(combinacoes,solucao_possivel);
    end
      imprime(sokucao_final,tamanho_solucao_final);
end
```

2.4.1 Avaliação do Algoritmo

Toda a verificação se o subfigure de vértices é ou não uma solução possível é feita para todas as combinações possíveis entre os vértices do grafo em questão. Nesse caso, temos que o número de soluções a serem avaliadas é 2^n, uma vez que o conjunto potência de um conjunto é 2 elevado ao número de elementos do conjunto [44]. Assim podemos dizer que a complexidade de tempo dessa solução implementada é $O(2^n)$.

Fazendo uma avaliação mais cuidadosa do algoritmo implementado, temos que a função que verifica se um determinado conjunto de vértices é ou não uma solução

possível, testa todos os pares possíveis do conjunto se existe uma aresta entre os mesmos. No entanto a partir do momento em que ela encontra uma aresta, ela aborta a restante da verificação. Assim, podemos dizer que, para] nosso algoritmo, quanto mais arestas existirem no grafo, mais rápida é a execução. Da mesma forma, temos que um grafo completamente desconexo é o pior caso para nosso algoritmo.

Dessa forma, no intuito de verificar qual o maior problema que o algoritmo em questão consegue resolver, criamos vários arquivos de entrada de grafos completamente desconexos, ou seja, o conjunto de vértices independentes do grafo é o próprio grafo, e executamos nosso programa variando o tamanho da entrada entre 1 e 30, medindo o tempos de usuário para cada uma dessas entradas. O resultado desses experimentos pode ser observado na Tabela 3.

Tabela 3: Relação Tempo de Execução pelo Tamanho do Grafo

Tamanho do Grafo	Tempo Execução(s)
1	0.000000
5	0.000000
10	0.000000
11	0.000999
12	0.001999
13	0.002999
14	0.006998
15	0.014997
16	0.031995
17	0.066989
18	0.147977
19	0.315951
20	0.684895
21	1.471776
22	3.154520
23	6.836960
24	14.510794
25	30.921299
26	66.003965
27	140.545633
28	296.465930
29	634.406555
30	1341.287093

Foi feita uma aproximação desses dados com a curva $f(x) = a^{x-b}$ através de uma regressão utilizando a ferramenta Gnuplot [17]. Dessa forma encontramos a função $f(x) = 2.12125^{x-20.4233}$, ou seja, comprovamos a complexidade de tempo $O(2^n)$. O gráfico dos dados medidos e da curva encontrada pode ser observado na Figura 11 e em escala logarítmica na Figura 12, mostrando a convergência das mesmas, que é praticamente perfeita.

Observando os resultados medidos através dos experimentos realizados, pode-

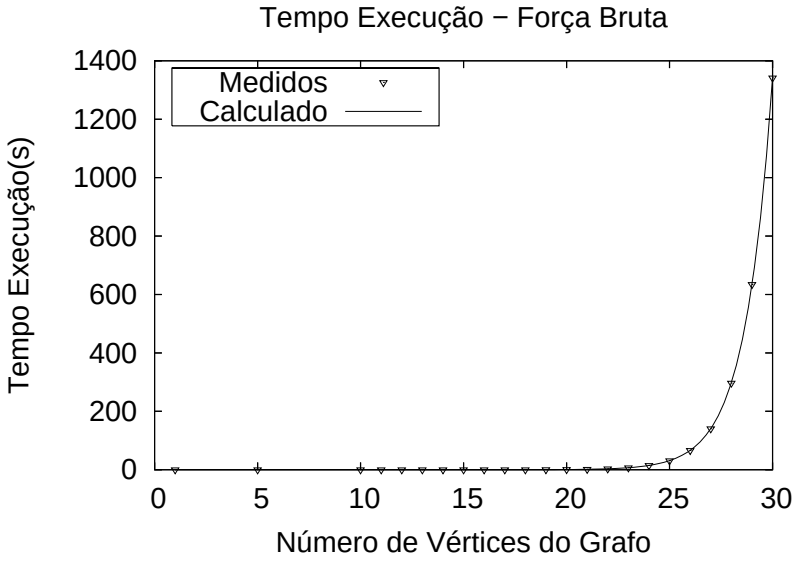

Figura 11: Análise de Complexidade - Solução Ótima

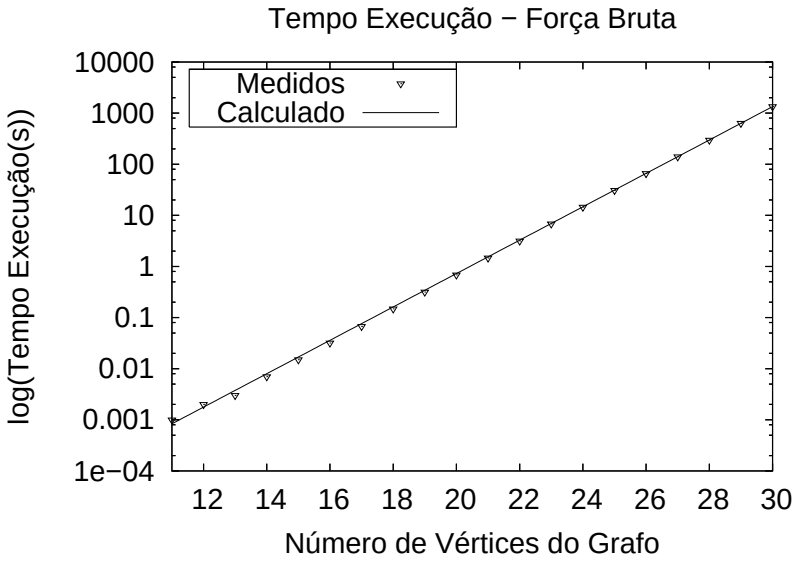

Figura 12: Análise de Complexidade - Solução Ótima(Escala Logarítmica)

mos perceber que já para uma entrada de 30 vértices completamente desconexos, nosso algoritmo executou em aproximadamente 1402s, o que corresponde a 23 minutos aproximadamente, mas no que no entanto era esperado, dada a ordem de complexidade calculada. Dessa forma, utilizando a função obtida através da regressão dos dados medidos($f(x) = 2.12125^{x-20.4233}$), temos que o tempo gasto para que o algoritmo em questão processe um grafo com um tamanho 10 vezes maior, ou seja, 300 vértices, seria de aproximadamente $2.77238260356273e + 91s$, ou seja, 8.84938081073336e+83 anos!!!

2.4.2 Exemplo de Funcionamento

Para mostrar o funcionamento correto do algoritmo que implementamos,apresentamos abaixo dois exemplos de grafos e o resultado obtido executando o algoritmo sobre esses dois grafos.

O primeiro deles é o próprio grafo dado de exemplo na questão, mostrado na Figura 13.

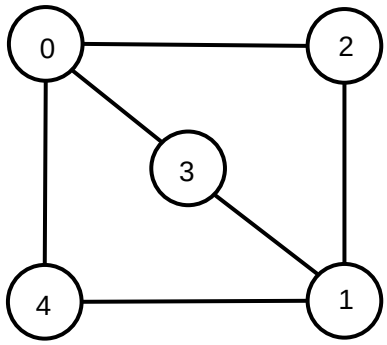

Figura 13: Grafo para Teste

Para esse grafo, obtivemos a seguinte resposta do algoritmo:

```
O No máximo de Nvertices independentes é 3.
São eles: 2 3 4
```

O segundo deles é uma versão do grafo dado de exemplo na questão só que completamente conectado, mostrado na Figura 14.

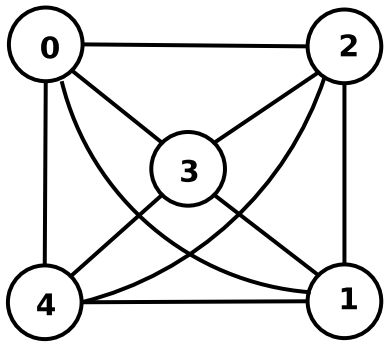

Figura 14: Grafo Completo

Para esse grafo, obtivemos a seguinte resposta do algoritmo:

```
O No máximo de vértices independentes é 1.
São eles: 0
```

2.5 Redução Polinomial ao Problema do Clique

O problema do conjunto de vértices independentes máximo de um grafo pode ser polinomialmente reduzido ao problema do clique. Essa redução é feita por meio da inversão da arestas, ou seja, onde existe aresta no grafo, a mesma deixa de existir e onde não existe aresta, a mesma passa a existir. Em nossa implementação, criamos uma função que inverte toda a matriz de adjacência e para executar o algoritmo para resolver o problema do clique, basta na linha de execução passar o parâmetro *-c*. Um pseudo-código para esse algoritmo pode ser visto abaixo:

```
void forca_bruta()
begin
    tamanho_solucao_final = -1;
    le_arquivo(matriz,Nvertices);
    inverte_matriz(matriz,Nvertices);
    gera_todas_combinacoes(combinacaoes,Nvertices);
    retorno = obtem_combinacao(combinacoes,solucao_possivel);
    while (retorno>=0)
    begin
      valida = testaPossibilidade(solucao_possivel,tamanho_solucao);
      if (valida){
          if (tamanho_solucao>=tamanho_solucao_final){
              solucao_final = solucao_possivel;
              tamanho_solucao_final = tamanho_solucao;
          }
      }
      retorno = obtem_combinacao(combinacoes,solucao_possivel);
    end
      imprime(sokucao_final,tamanho_solucao_final);
end
```

Assim, feita essa redução, ao aplicarmos nosso algoritmo, a saída do mesmo é o conjunto de vértices que forma o maior clique do grafo. Nesse caso, retomando a avaliação cuidadosa que fizemos do algoritmo implementado, temos que a função que verifica se um determinado conjunto de vértices é ou não uma solução possível, testa todos os pares possíveis do conjunto se existe uma aresta entre os mesmos. No entanto a partir do momento em que ela encontra uma aresta, ela aborta a restante da verificação. Assim, podemos dizer que, para nosso algoritmo resolva o problema do clique, quanto mais arestas existirem no grafo, como faremos a transformação polinomial invertendo as arestas, mais lenta é a execução. Da mesma forma, temos que um grafo completamente desconexo é o melhor caso para aplicação do nosso algoritmo com a redução polinomial, o que é bastante coerente com a análise já feita anteriormente.

Assim temos que toda a análise de complexidade de tempo apresentada na seção 2.4.1 para o nosso algoritmo que calcula o conjunto de vértices independentes é válida para o uso do mesmo para o cálculo do clique máximo do grafo. A única diferença é que ao invés de utilizarmos vértices completamente desconexos, devemos

utilizar grafos completamente conexos para calcular o maior problema de clique que o algoritmo consegue resolver.

2.5.1 Exemplo de Funcionamento

Para mostrar o funcionamento correto do algoritmo que implementamos para o problema do clique, apresentamos abaixo dois exemplos de grafos e o resultado obtido executando o algoritmo sobre esses dois grafos.

O primeiro deles é o próprio grafo dado de exemplo na questão, mostrado na Figura 15.

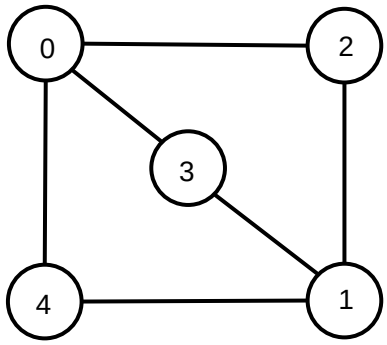

Figura 15: Grafo para Teste

Para esse grafo, obtivemos a seguinte resposta do algoritmo:

```
O tamanho do clique é 2.
São os vértices: 0 2
```

O segundo deles é uma versão do grafo dado de exemplo na questão só que completamente conectado, mostrado na Figura 20.

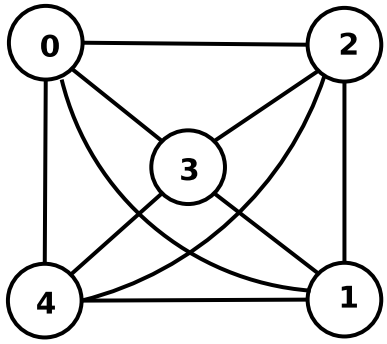

Figura 16: Grafo Completo

Para esse grafo, obtivemos a seguinte resposta do algoritmo:

```
O tamanho do clique é 5.
São os vértices: 0 1 2 3 4
```

2.5.2 Algumas Considerações

A solução apresentada para obter a solução ótima para o problema do conjunto de vértices independentes, assim como para o problema do clique máximo de um grafo(através da redução polinomial) é uma solução bastante simples e conseqüentemente não eficiente. Temos que algumas podas poderiam ser feitas pelo algoritmo a medida em que determinados subconjuntos fosse tidos como não pertencentes a solução final. No entanto optamos por essa implementação para mostrar a natureza exponencial do problema e suas consequências. Assim, a seguir apresentamos um algoritmo que gera a solução ótima para tais problemas em tempo médio de execução bem melhor que o apresentado, apesar de manter a característica exponencial, ou sejam no pior caso ainda ser $O(2^n)$.

2.6 Uma Solução Ótima Alternativa

Como o algoritmo que obtém a solução ótima implementado se mostrou bastante inficiente em relação ao tempo de execução, foi elaborado juntamente com o aluno Bruno Rocha Coutinho, uma solução alternativa que também obtém a solução ótima que no pior caso também é $O(2^n)$, mas que na prática consegue ser bem melhor. Esse algoritmo é baseada em um algoritmo de mineração de dados que procura itemsets frequentes em base de dados, o GenMax [21]. A idéia de implementação desse algoritmo surgiu graças a experiência de ambos os alunos com técnicas de mineração de dados de onde foi possível modelar o problema em questão para um problema de mineração de dados.

Dada uma base de transaçãoes $T = \{t_{1,}, t_2, \ldots, t_n\}$, como o registro de compras de uma loja de conveniência, por exemplo, deseja-se determinar conjuntos de produtos que são muitas vezes comprados juntos, pois nesse caso a compra de um produto pode implicar na compra de outros. Assim temos que um *itemsets* é um conjunto $I = \{i_1, i_2, \ldots, i_k\}$ de k ítens distintos, o qual é considerado freqüente se aparece na base pelo menos m vezes, onde m é uma constante arbitrária. Esse itemset é considerado maximal, se não existe itemset freqüente que o contenha.

O genMax é um algoritmo baseado em backtrack para minerar os *itemsets* maximais de uma lista de transações, o qual utiliza várias otimizações para podar o espaço de busca e assim reduzir consideravelmente o tempo de execução.

As propriedade para ambos os problemas são muito parecidas: Se um conjunto atende à restrição (é frequente no caso dos) itemsets maximais, todos os seus subconjuntos devem atender à restrição, já no problema do maior conjunto de vértices independentes, a restrição é que todos os vértices do conjunto não devem se conectar. Dados um conjunto C de vértices onde nenhum dos seus elementos se conectam, se retirarmos qualquer elemento de C os vértices continuarão não se conectando; Se um conjunto C não atende à restrição, todos os conjuntos que contém C também não atendem à restrição, se um conjuto possui 2 vértices que se conectam, ele não atende à restrição de que todos os vértices não podem se conectar e não importa quantos vértices adicionemos ao conjunto, ele continuará a ter pelomenos 2 vértices que se conectam.

Assim, em nosso caso, queremos encontrar o maior conjunto maximal de vértices independentes. Assim a adaptação do GenMax para o nosso problema consiste basicamente em alterar a função que verifica se um determinado itemset é frequente

em uma base de dados para que a mesma verifique se existe uma aresta entre os membros de cada conjunto maximal encontrado na matriz de adjacência.

Apesar de elaborarmos juntos a solução utilizando o GenMax e realizar parte da implementação juntos desse problema, o aluno Bruno Rocha Coutinho foi quem finalizou a implementação, assim não entraremos e maiores detalhes da implementação desse algoritmo nessa documentação. No entanto, como os tempos de execução para o força bruta implementado foi muito ruim, utilizamos esse algoritmo para gerar resultados que pudessem ser comparados com as demais implementações realizadas nesse trabalho, tanto em questão de tempo de execução quanto em questão da solução ótima obtida por ele.

2.6.1 Resultados Experimentais

Apenas como uma ilustração do quão bom esse algoritmo pode ser se comparado com um algoritmo de força bruta, apresentamos a seguir alguns gráficos que comparam o tempo de execução de ambos para conjunto de grafos de características diferentes. Foram gerados grafos aleatórios com grau de conectividade variando entre 1 e 3 e a quantidade de vértices variando entre 10 e 70. Para cada um desses grafos foram feitas 5 execuções e o tempo de execução final foi feito com base na média dos tempos de execução encontrados em cada uma das execuções.

No gráfico da Figura 17 apresentamos os resultados de tempo de execução para grafos com conectividade 1.

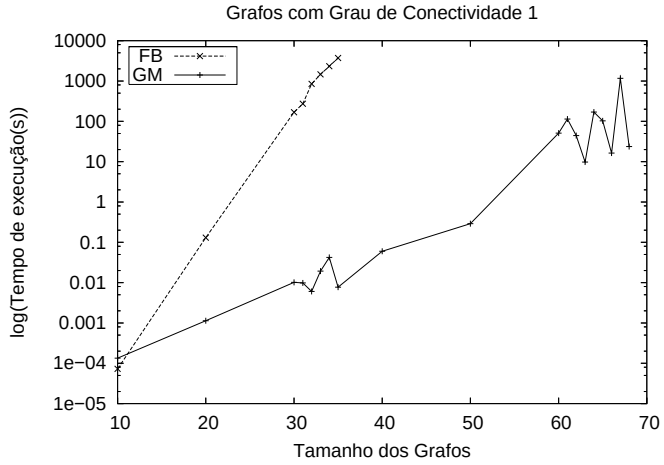

Figura 17: Tempo de Execução - Força Bruta x GenMax

No gráfico da Figura 18 apresentamos os resultados de tempo de execução para grafos com conectividade 2 e por fim no gráfico da Figura 19 apresentamos os resultados de tempo de execução para grafos com conectividade 3.

Em todos os gráficos apresentados acima podemos perceber que o tempo de execução do algoritmo GenMax foi bem melhor do que o algoritmo força bruta, mesmo ambos tendo complexidade $O(2^n)$. Isso acontece porque o algoritmo GenMax aplica várias podas durante seu processamento, o que, como podemos ver, melhora significativamente o tempo de execução. Outro fato interessante de se observar, que ambos algoritmo apresentam um a melhora a medida em que aumentamos a

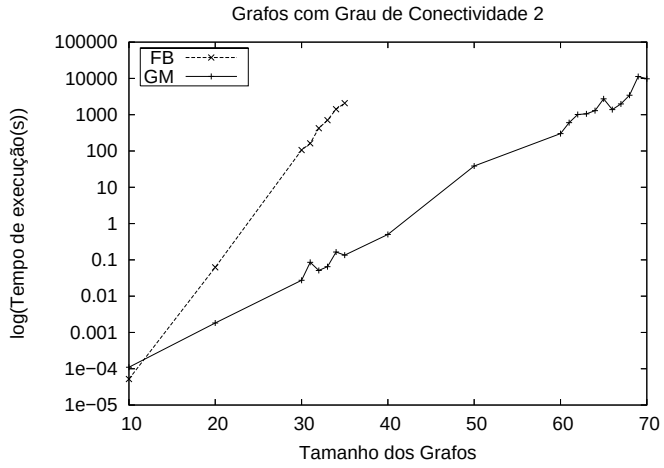

Figura 18: Tempo de Execução - Força Bruta x GenMax

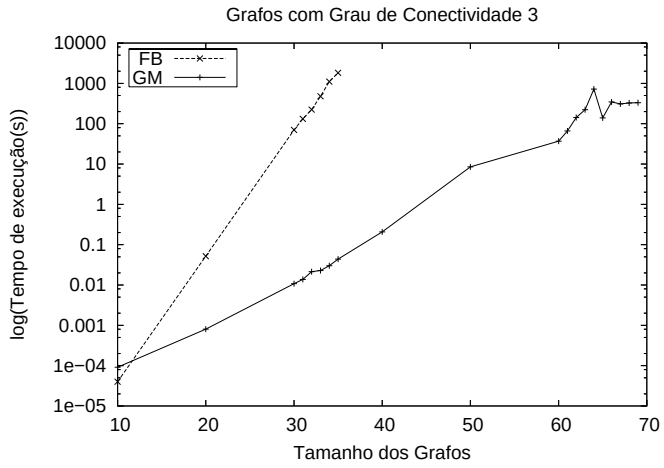

Figura 19: Tempo de Execução - Força Bruta x GenMax

conectividade dos grafos. No algoritmo de força bruto, isso é explicado pelo fato de que, na avaliação de uma solução se a mesma é ou não válida, assim que a função encontra uma aresta, ela já retorna sem analisar o restante do conjunto, e como em grafos mais conectados a quantidade de arestas é maior, faz com que a probabilidade de se encontrar uma aresta em um conjunto qualquer seja maior, e com não é preciso fazer a verificação para os demais. Já para o algoritmo GenMax essa melhora se deve ao fato que o mesmo possui várias podas e a prpbabilidade de uma poda acontecer aumenta a medida que a conectividade do grafo aumenta, pois mais arestas teremos.

2.7 Alguns Algoritmos Interessantes Descritos na Literatura

O problema do conjunto máximo de vértices independentes, assim como o problema do maior clique e de coloração de grafo são problemas já bastante estudados na literatura. No intuito de entender melhor o problema e as diversas soluções de heurísticas propostas, apresentamos nessa alguns dos principais artigos nessa área. A

partir desse estudo, selecionamos algumas que forma implementadas nesse trabalho.

No artigo [10] os autores apresentam dois algoritmos baseados em *backtracking* que utilizando técnicas de *branchand-bound* para podar aquelas soluções parciais que não levar a um clique. A primeira versão é uma implementação direta de um algoritmo básico do problema. Esta implementação foi utilizada para ilustrar o método utilizado, o qual gera cliques em ordem lexicográfica. A segunda versão é derivada da primeira e gera primeiramente grandes cliques e então fera cliques seqüencialmente tendo uma larga intersecção.

O artigo [9] descreve métodos heurísticos novos e eficientes para coloração de grafos baseados em comparação de grau e estrutura de grafos. Um método desenvolvido para grafos bipartidos é uma importante parte para os procedimento heurísticos para encontrar o clique máximo em grafos gerais. Além desse, existem outros diversos trabalhos sobre coloração de grafos que podem muito bem ser adaptados ao problema do conjunto máximo de vértices independentes como [34, 4, 29, 28]

Em [8] os autores apresentam uma heurística baseada em recursão para resolver o problema do máximo conjunto de vértices independentes. Primeiramente, para o gráfico todo, escolhe-se um pivô aleatoriamente e a partir do mesmo seleciona-se todos os vizinhos e não vizinhos do mesmo e isso é feito recursivamente para esses dois novos conjuntos até que não existam mais aresta no grafo. Cada chamada recursiva retorna o clique máximo encontrado e o conjunto de vértices independentes máximo encontrado. A partir desses resultado, escolhe-se qual retornou o maior clique e o maior conjunto independente e retorna para a chamada anterior, até que todas as chamados seja retornadas e tem-se então o conjunto de vértices independentes máximo. Além disso, os autores sugerem a aplicação de uma customização, baseada na remoção de cliques para se obter uma resposta ainda melhor.

Em [12] os autores propõem uma heurística para o problema do do máximo conjunto de vértices independentes que utiliza resultados para o problema da função de otimização quadrática sobre esferas. Essa proposta mostrou-se bastante eficiente através de experimentos executados com os *benchmarks* do DIMACS. Em [45] os autores apresenta um método baseado em *branch-and-price* (B&P) para resolver o problema de conjunto independente de vértices de peso máximo (MWISP). Nesse método o grafo e particionado em conjunto menores nos quais o problema do (MWISP) é mais fácil de se resolver O resultados desses sub-grafos são repassados para um arcabouço B&P que resolve o problema original do (MWISP). Os resultados desse artigos são bastante interessantes para grafos esparsos

Em [39] são apresentados diversas heurísticas para o problema do conjunto máximo de vértices independentes e para o problema do maior clique. Nesse artigo encontramos algumas sugestões utilizando algoritmos gulosos como [26]. Encontramos também algumas propostas de heurísticas de busca local, além de vários exemplos de heurísticas mais avançadas como *Simulated Anealing*, redes neurais, algoritmos genéticos [6], entre outros.

Dessa forma, optamos pela implementação de três heurísticas diferentes: uma implementação gulosa, baseada na escolha da melhor solucao do momento; a heurísticas apresentada por [8], brevemente acima; e por uma heurística mais avançada que utiliza algoritmos genéticos. Cada uma dessas heurísticas é apresentada abaixo, além da avaliação das mesmas.

2.8 Solução Gulosa

Nesta seção apresentamos o algoritmo que implementamos que se baseia na melhor solução no momento, ou seja, podemos dizer que se trata de uma heurística gulosa. Assim como no algoritmo de solução ótima, primeiramente é feita a leitura do arquivo de entrada que contém o grafo. Nesse arquivo de entrada, temos na primeira linha o número de vértices do grafo e nas linhas posteriores temos um par de números (vértices) que representavam um aresta entre eles. Assim é feita a leitura do arquivo e uma matriz de adjacência de tamanho (número de vértices) X (número de vértices) é criada. Nessa matriz armazenamos as arestas de cada vértice.

Feita a leitura do grafo e armazenada a matriz de adjacência, o próximo passo do algoritmo é ordenar os vértices de acordo com a quantidade de arestas que o mesmo possui, ou seja, realizamos uma ordenação do vértice menos conectado para o vértice mais conectado. Para realizar essa ordenação utilizamos o um algoritmo de ordenação de menor ordem de complexidade, o Heapsort [48], cujo comportamento é $O(n \log n)$.

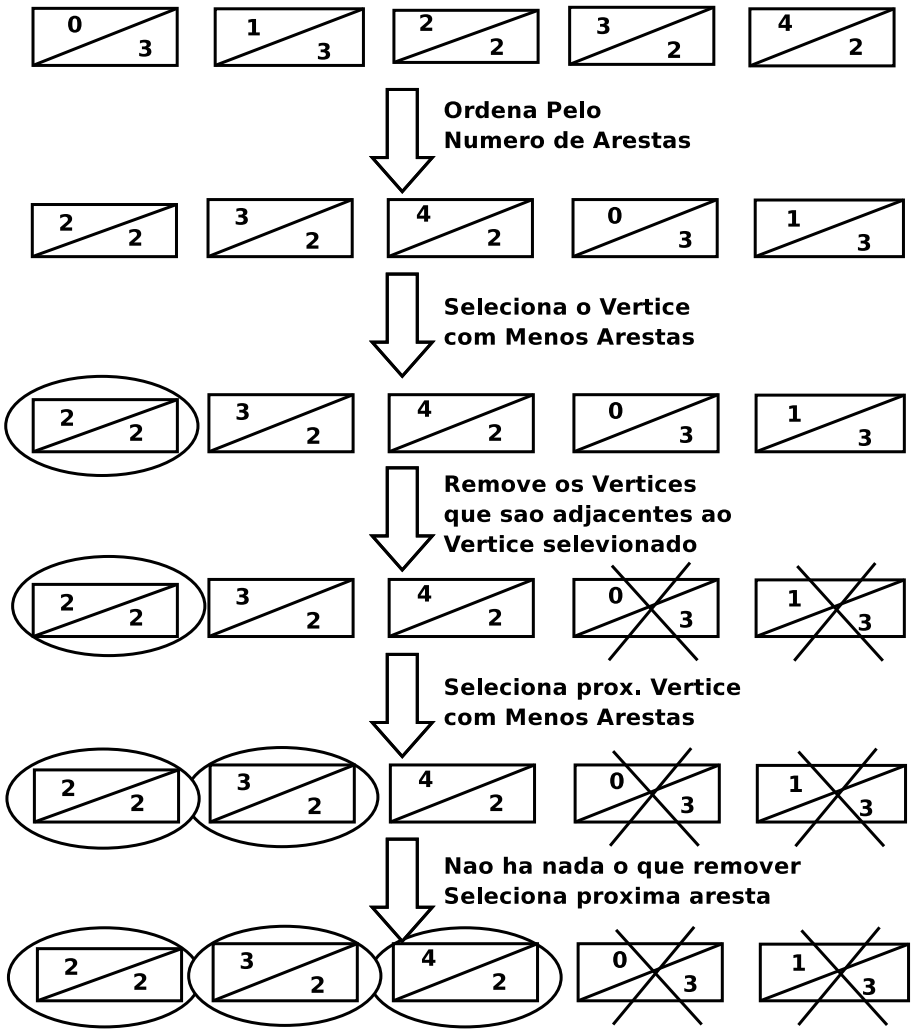

Figura 20: Funcionamento Guloso

Nesse caso, criamos uma estrutura denominada *Item* que contém a quantidade

de arestas do vértice e o número do próprio vértice. Essa estrutura contendo todos os vértices do grafo é passada para o procedimento de ordenação Heapsort, que faz a ordenação pela quantidade de arestas.

Feita a ordenação dos vértices, o vértice menos conectado é inserido na solução final do problema e todos os vértices que possuem alguma aresta com esse vértice são descartados pelo algoritmo a partir desse momento. O próximo vértice menos conectado e que não foi descartado é então inserido na solução final do problema. Novamente todos os vértices que possuem alguma aresta com esse vértice são descartados pelo algoritmo a partir desse momento. Isso é feito repetidas vezes até que não exista mais nenhum vértice possível de ser inserido na solução final. Na Figura 20 podemos ver um diagrama de funcionamento do nosso algoritmo guloso para o grafo da Figura 15 e abaixo segue um pseudo-código para o mesmo.

```
void vertices_independentes_guloso()
begin
    Item **matriz_adjacencia;
    File *arq;
    Item *Vetor;
    int *solucao
    int i = 0;
    int vertice = 0;
    leitura_arquivo_entrada(arq,matriz_adjacencia);
    Conta_arestas_por_vertice(matriz_adjacencia,Vetor);
    Heapsort(Vetor);
    vertice = Obtem_proximo_vertice_menos conectado(Vertor);

    while(vertice>=0)
    begin
        solucao[i] = vertice;
        Remove_vertices_adjacentes(vertice,Vetor);
        i++;
        vertice = Obtem_proximo_vertice_menos conectado(Vertor);
    end
    imprime(Vetor);
end
```

2.8.1 Avaliação do Algoritmo

Analisando o pseudo-código apresentado acima para nosso algoritmo guloso temos que a função que conta o número de arestas por vértice varre toda a matriz de adjacência para contabilizar a quantidade de arestas, dessa forma é um procedimento com complexidade $O(n^2)$. Temos que o Heapsort possui ordem de complexidade $O(n \log n)$ como apresentado em [48].

Além desses dois procedimentos, temos a função que obtém o próximo vértice menos conectado e a função que remove todos os vertices adjacentes a um determinado vértice. Na função que obtém o o próximo vértice menos conectado é feita uma procura pelos N vértices do grafo pelo de menor número de arestas que ainda

não foi inserido na solução, assim, no pior caso temos que esse procedimento tem complexidade $O(n)$. Na função que remove os vértices adjacentes, no pior caso também percorre todos os N vértices verificando se o mesmo possui ou não aresta com o vértice em questão, dessa forma esse procedimento também possui complexidade $O(n)$. Ambos os procedimentos estão inseridos em uma estrutura de repetição a qual, no pior caso (um grafo completamente desconexo), pode executada N vezes, onde N é o número de vértices do grafo. Dessa forma, temos que esse loop tem uma complexidade de $O(n^2) + O(n^2)$, o que é portanto $O(n^2)$.

Assim, temos que a complexidade do nosso algoritmo é dada por:

$$MAX(O(n^2), O(n \log n), O(n^2))$$

Ou seja, $O(n^2)$. Dessa forma, no intuito de verificar a ordem de complexidade de tempo do algoritmo em questão, criamos vários arquivos de entrada de grafos completamente desconexos, ou seja, o conjunto de vértices independentes do grafo é o próprio grafo, e executamos nosso programa variando o tamanho da entrada entre 10 e 10000, medindo o tempos de usuário para cada uma dessas entradas. O resultado desses experimentos pode ser observado na Tabela 4. Os grafos escolhidos foram desconexos pois, para o algoritmo em questão, representa a pior tipo de entrada possível, pois todos os vértices serão sempre avaliados a cada momento.

Foi feita uma aproximação desses dados com a curva $f(x) = a(x^2)$ através de uma regressão utilizando a ferramenta Gnuplot [17]. Dessa forma encontramos a função $f(x) = 8.17596e - 09(x^2)$, ou seja, comprovamos a complexidade de tempo $O(n^2)$. O gráfico dos dados medidos e da curva encontrada pode ser observado na Figura 30 mostrando a convergência das mesmas, que é praticamente perfeita.

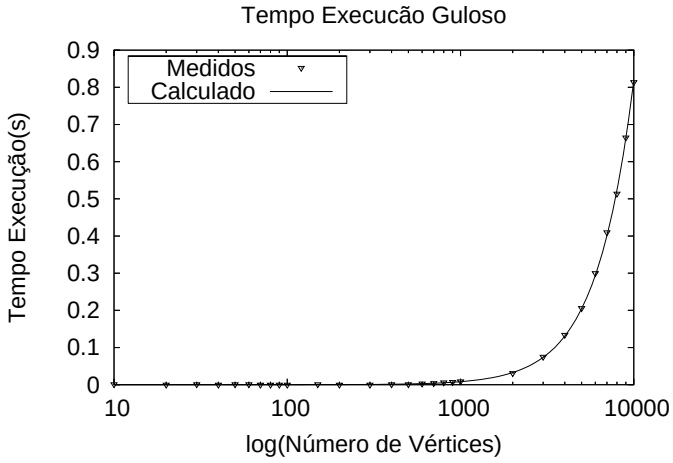

Figura 21: Análise Complexidade de Tempo - Guloso

Tabela 4: Relação Tempo de Execução pelo Tamanho do Grafo - Solução Gulosa

Tamanho do Grafo	Tempo Execução(s)
10	0.000999
20	0.000000
30	0.000999
40	0.000000
50	0.000999
60	0.000999
70	0.000000
80	0.000000
90	0.000000
100	0.000000
150	0.000999
200	0.000000
300	0.000000
400	0.000999
500	0.000999
600	0.002999
700	0.003999
800	0.005999
900	0.006998
1000	0.008998
2000	0.030995
3000	0.074988
4000	0.133979
5000	0.205968
6000	0.299954
7000	0.409937
8000	0.513921
9000	0.664898
10000	0.813876

2.8.2 Exemplo de Funcionamento

Para mostrar o funcionamento correto do algoritmo que implementamos,apresentamos abaixo dois exemplos de grafos e o resultado obtido executando o algoritmo sobre esses dois grafos.

O primeiro deles é o próprio grafo dado de exemplo na questão, mostrado na Figura 13. Para esse grafo, obtivemos a seguinte resposta do algoritmo:

```
O No máximo de vertices independentes é 3.
São eles: 2 3 4
```

O segundo deles é uma versão do grafo dado de exemplo na questão só que completamente conectado, mostrado na Figura 14. Para esse grafo, obtivemos a

seguinte resposta do algoritmo:

```
0 No máximo de vertices independentes é 1.
São eles: 0
```

2.8.3 Avaliação das Soluções

Nessa seção apresentamos uma avaliação da qualidade das resposta obtidas executando nosso algoritmo guloso para o problema do conjunto máximo de vértices independentes. Para esse algoritmo não encontramos uma aproximação analítica que nos desce o quão próxima as soluções encontradas pelo mesmo está da solução ótima, no entanto, como apresentado na seção 2.3, sabemos que a mesma não pode ser um fator de $2^{\frac{\log\log n}{\log\log\log n}}$, caso contrário P=NP. Além disso, temos que encontrar pode uma tarefa extremamente complicada, uma vez em [7] argumenta-se que esse problema não pode ser aproximado com nada a menos que seja limitado por um n.

Dessa forma, apresentamos a qualidade das respostas do algoritmo guloso medidas empiricamente. Para isso, foram gerados grafos aleatórios com grau de conectividade variando entre 1 e 3 e a quantidade de vértices variando entre 10 e 70. Para cada um desses grafos foram feitas 5 execuções e a solução final foi feita com base na média das soluções encontradas em cada uma das execuções.

O primeiro desses resultados pode ser observado no gráfico na Figura 22 onde apresentamos a qualidade das soluções para os grafos de conectividade 1. Para as medições realizadas temos que a qualidade da respostas obtidas pela solução gulosa nunca foi inferior a 90%, mesmo assim nada podemos afirmar com relação a sua aproximação, no entanto mostra que o mesmo possui uma qualidade muito boa de suas soluções para grafos com conectividade baixa e que pode ser explicada pela forma com que o algoritmo trabalha. O algoritmo ordena os vértices pela quantidade de arestas que o mesmo possui e sempre coloca na solução os menos conectados, e como a quantidade de arestas por vértice é muito próxima, a tendência é que tenhamos uma boa solução.

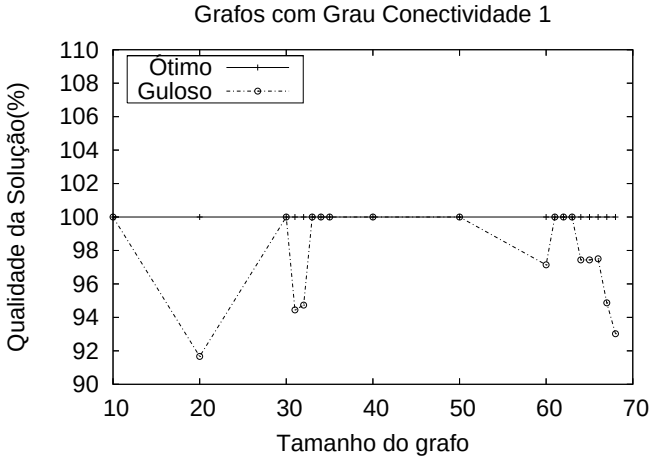

Figura 22: Qualidade da Solução - Guloso

No gráfico da Figura 23 apresentamos a qualidade das soluções para os grafos de conectividade 2. Para as medições realizadas temos que a qualidade da respostas obtidas pela solução gulosa nunca foi inferior a 85%, mesmo assim nada podemos afirmar com relação a sua aproximação. Apesar da qualidade das soluções obtidas pelo algoritmo para essa classe de grafos também se mostram muito boas, se compararmos com a qualidade apresentadas para grafos de conectividade 1, observamos um pequeno declínio. Mais uma vez isso pode ser explicado pelo funcionamento do algoritmo. O algoritmo ordena os vértices pela quantidade de arestas que o mesmo possui e sempre coloca na solução os menos conectados, e como a quantidade de arestas por vértice é muito próxima e, nesse caso, começa a aumentar, a tendência é que tenhamos uma solução não tão boa.

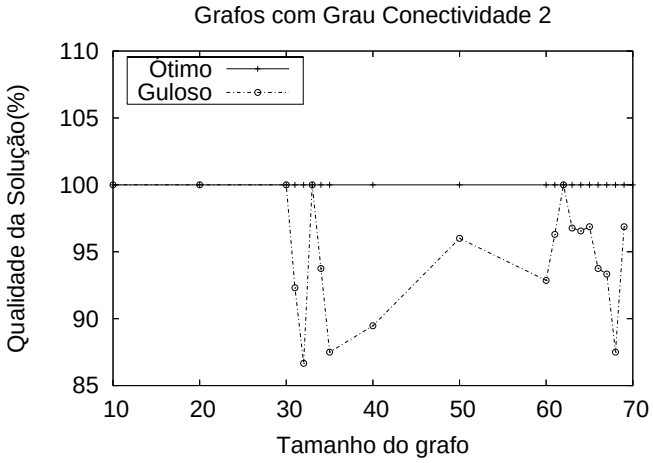

Figura 23: Qualidade da Solução - Guloso

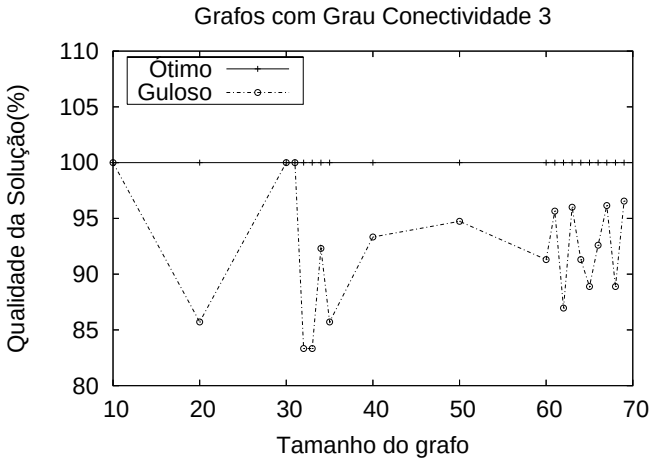

Figura 24: Qualidade da Solução - Guloso

Por fim, no gráfico da Figura 24 apresentamos a qualidade das soluções para os grafos de conectividade 3. Para as medições realizadas temos que a qualidade da respostas obtidas pela solução gulosa nunca foi inferior a 80%, mesmo assim nada

podemos afirmar com relação a sua aproximação. Apesar da qualidade das soluções obtidas pelo algoritmo para essa classe de grafos também se mostram muito boas, se compararmos com as qualidades apresentadas para grafos de conectividade 1 e 2, novamente observamos um pequeno declínio. Mais uma vez isso pode ser explicado pelo funcionamento do algoritmo. O algoritmo ordena os vértices pela quantidade de arestas que o mesmo possui e sempre coloca na solução os menos conectados, e como a quantidade de arestas por vértice é muito próxima e, nesse caso, começa a aumentar, a tendência é que tenhamos uma solução não tão boa.

2.9 Algoritmo por Remoção de Cliques

Nessa seção apresentamos nossa implementação da solução proposta por [8]. Essa uma heurística que utiliza recursão para resolver o problema do máximo conjunto de vértices independentes. Essa heurística se baseia em uma estratégia gulosa. Escolhendo um vértices qualquer v para fazer parte da nossa solução. O restante da solução deve então ser preenchida com o conjunto dos vértices não-vizinhos de v ($\overline{N}(v)$). Especificando esse resultado formalmente temos:

$$Escolha \quad v \in V(G)$$
$$I(G) \longleftarrow v \cup I(\overline{N}(v))$$

Essa solução é bastante atraente pois encontra de forma bastante rápida a solução para o problema. No entanto, se desconsiderássemos o vértice v, poderíamos encontrar uma outra solução com mais vértices que a encontrada. Além disso, se a escolha desse pivô for feita de forma equivocada, ou seja, escolhêssemos um vértice muito conectado, obteríamos um conjunto de vértices independentes muito pequeno, ou seja, temos uma solução muito ruim no pior caso.

Dessa forma, vamos tomar uma nova regra para obtermos o conjunto independente. Como anteriormente, escolhemos um vértice construímos o conjunto dos vértices não-vizinhos a esse pivô. Mas ao mesmo tempos construímos o conjunto de vértices vizinhos a esse pivô e então escolhemos aquele que obtiver o melhor resultado. Da mesma forma, essas regras se aplicam a conjunto clique. Assim temos mais formalmente:

$$Escolha \quad v \in V(G)$$
$$I(G) \longleftarrow MAX(v \cup I(\overline{N}(v)), I(N(v)))$$
$$C(G) \longleftarrow MAX(v \cup C(\overline{N}(v)), C(N(v)))$$

Assim temos como resultado para essas definições o seguinte algoritmo:

```
solucao GeraSolucao(G)
begin
    se vazio(G) {
        retorna (vazio,vazio);
    }
    EscolhaVertice(G);
    V  = Vizinhos(G,v);
    Nv = Nvizihos(G,v);
    (C1,I1) = GeraSolucao(V);
    (C2,I2) = GeraSolucao(Nv);
    retorna(MAX((C1 uniao v),C2),MAX((I2 uniao v),I1));
end
```

Observando o algoritmo, temos que primeiramente, para o gráfico todo, escolhe-se um pivô aleatoriamente e a partir do mesmo seleciona-se todos os vizinhos e não vizinhos do mesmo e a a função é chamada recursivamente para esses dois novos conjuntos até que não existam mais aresta no grafo. Cada chamada recursiva retorna o clique máximo encontrado e o conjunto de vértices independentes máximo encontrado. A partir desses resultado, escolhe-se qual retornou o maior clique e o maior conjunto independente e retorna para a chamada anterior, até que todas as chamados seja retornadas e tem-se então o conjunto de vértices independentes máximo.

Sobre esse algoritmo, existe ainda uma customização que pode ser feita baseada na remoção de cliques para se obter uma resposta ainda melhor para o conjunto máximo de vértices independentes. Isso é feito sobre o princípio de que cada conjunto de vértices independentes e o conjunto dos vértices do clique possuem no máximo um vértice em comum. Assim, de forma interativa, a função GeraSolucao é chamada várias vezes, e para cada nova chamada da função, removemos do grafo a ser passado para a função todos os vértices encontrados no conjunto clique anterior. Isso é feito até que todos os vértices do grafo seja removido. A cada iteração, verifica-se se o conjunto encontrado é maior que o maior conjunto até aquele momento, em caso afirmativo o mesmo é substituído. Segue abaixo um pseudo-código para essa solução, utilizando o código já apresentado acima.

```
Remove_Clique(G)
begin
    conjunto_maximo = vazio;
    enquanto(!vazio(G)) faca
    begin
        (C,I) = GeraSolucao(G);
        se (I > conjunto_maximo){
            conjunto_maximo = I;
        }
        RetiraClique(C,G);
    end
end
```

2.9.1 Avaliação do Algoritmo

Para realizar a avaliação de complexidade dessa solução implementada, vamos primeiramente analisar a complexidade da função que gera solução, ou seja, o algoritmo de Ramsey. Nessa função, temos duas chamadas recursivas, uma para o conjunto de vértices vizinhos do pivô escolhido e uma para o conjunto de vértices não vizinhos. Dessa forma, temos que uma delas irá percorrer até todas as arestas e a outra poderá percorrer até todos os vértices. Isso nos dá uma ordem de complexidade de $O(|E|) + O(|V|)$, onde E são as arestas do gráfico e V os vértices do mesmo. Como a número de arestas na grande maioria dos casos é bem maior que o de arestas, temos que $O(|E|)$ domina assintoticamente a função, temos assim uma complexidade para o algoritmo de Ramsey de $O(|E|)$, coerente como a complexidade para o mesmo algoritmo apresentado em [38]. Em nossa implementação, função de Ramsey, ou GeraSolucao, possui várias atribuições, cada uma delas $O(1)$. Nessa mesma função, no entanto, temos oito loops de interação, dois para obter os conjuntos de vizinhos e não vizinhos, quatro para obter a quantidade de vértices independentes e no clique de cada sub-solução e dois loops onde são feitas as cópias das melhores soluções. Cada uma dessas iterações tem complexidade $O(V)$, assim a complexidade dessa função para cada iteração é $MAX(O(V) + O(V) + O(V) + O(V) + O(V) + O(V) + O(V) + O(V))$ que nos dá uma complexidade de $O(V)$. Como essa função é chamada recursivamente $|E|$ vezes, temos a complexidade total para o algoritmo de Ramsey implementado é dada por $O(|E|.|V|)$ em nossa implementação.

Analisando agora o procedimento de remoção de clique, temos que a cada iteração um clique é removido do grafo, até que o todos os vértices do grafo sejam removidos. Por [38] temos que o número de cliques que são removidos durante todas as iterações é dada por $n \log n$, como para cada uma dessas remoções o algoritmo de ramsey é executado temos a complexidade $O(|E|.n \log n)$. Como em nossa implementação temos que o algoritmo de Ramsey tem complexidade igual a $O(|E|.|V|)$ devido a cópia das sub-soluções, temos que a complexidade final do algoritmo de remoção de clique é $O(|E|.|V|.n \log n)$ para nossa implementação.

Seguindo o mesmo raciocíno acima descrito, como análise, vamos tomar um caso extremo como um grafo completamente desconectado. No caso de um grafo completamente desconectado, o conjunto de vértices não vizinhos de um pivô qualquer, escolhido aleatoriamente, será o restante do grafo e o conjunto de vizinhos será vazio. Dessa forma, a cada chamada recursiva, apenas um vértice será removido para a próxima chamada, nesse caso teremos V chamadas recursivas para a função de Ramsey e como cada chamada tem um custo V, temos nesse caso uma complexidade $O(V^2)$. Feita a análise do algoritmo de Ramsey, vamos analisar agora o algoritmo de remoção de cliques que utiliza a função de Ramsey. Continuando o caso extremo acima de um grafo completamente desconectados, temos que a cada iteração, apenas um vértice será removido do grafo, uma vez que o tamanho máximo de um clique para um grafo completamente conectado é 1. Nesse caso serão necessárias V iterações. Como o algoritmo de Ramsey possui uma complexidade de $O(V^2)$ e o mesmo é executado V vezes, temos assim nesse caso uma complexidade de $O(V^3)$, que domina $O(|E|.|V|.n \log n)$.

Dessa forma, no intuito de verificar a ordem de complexidade de tempo do algoritmo em questão para esse caso extremo, criamos vários arquivos de entrada de

grafos completamente desconexos, ou seja, o conjunto de vértices independentes do grafo é o próprio grafo, e executamos nosso programa variando o tamanho da entrada entre 10 e 10000, medindo o tempos de usuário para cada uma dessas entradas. O resultado desses experimentos pode ser observado na Tabela 5.

Tabela 5: Relação Tempo de Execução pelo Tamanho do Grafo - Ramsey(Remoção de Clique)

Tamanho do Grafo	Tempo Execução(s)
10	0.000000
20	0.000999
30	0.000999
40	0.002999
50	0.003999
60	0.007998
70	0.011998
80	0.016997
90	0.022996
100	0.028995
200	0.194970
300	0.644901
400	1.529767
500	3.025540
600	5.315191
700	8.436717
800	12.577087
900	17.919275
1000	24.624256

Foi feita uma aproximação desses dados com a curva $f(x) = a(x^3)$ através de uma regressão utilizando a ferramenta Gnuplot [17]. Dessa forma encontramos a função $f(x) = 2.45975e - 08(x^3)$, ou seja, comprovamos a complexidade de tempo $O(n^3)$. O gráfico dos dados medidos e da curva encontrada pode ser observado na Figura 25 mostrando a convergência das mesmas, que é praticamente perfeita.

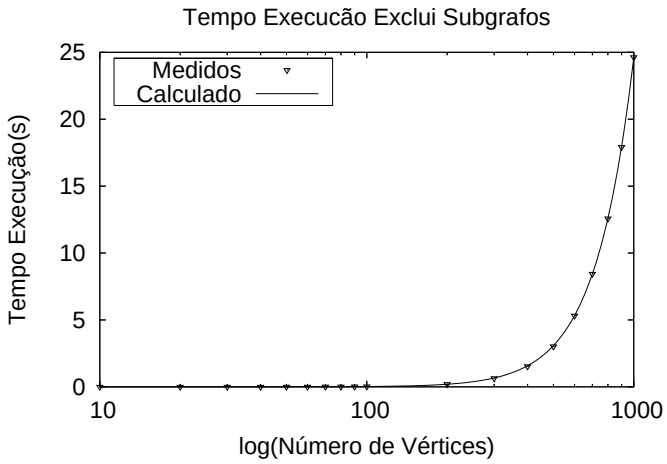

Figura 25: Análise Complexidade de Tempo - Ramsey(Remoção de Clique)

2.9.2 Aproximação do Algoritmo

Utilizando a teoria de Ramsey [47], Boppana and Halldórsson em [8] mostram que para o clique C e o conjunto independente I retornados pelo Ramsey(G) temos que $|C|.|I| \geq \frac{1}{4}(\log n)^2$. Este limite por si só não garante um tamanho mínimo para $|I|$ uma vez que $|C|$ pode ser bem grande. A proposta do algoritmos de exclusão de sub-grafo é feita para modificar o grafo uma vez que, $|C|$ pode ser muito pequeno. Isto é obtido pela chamada repetida de Ramsey e excluindo (removendo) o clique retornado.

O conjunto independente pode perder no máximo um vértice a cada iteração, isto porque o clique e o conjunto independente podem compartilhar no máximo um vértice. Se o grafo possui um conjunto suficientemente de vértices independentes, uma fração constante do grafo será mantida se todos os vértices do clique de um certo tamanho mínimo k são excluídos. Se o algoritmo de Ramsey é executado no grafo resultante, o tamanho de C será no máximo k. Isto implica que o limite inferior será $|I| \geq \frac{(\log n)^2}{4k}$. Se o maior conjunto independente é pequeno, o desempenho desse algoritmo será muito bom. O resultado dessa análise é um garantido de $O(\frac{n}{(\log n)^2})$. Maiores detalhes sobre como obter a aproximação aqui mostrada pode ser encontrada em [8, 25]

2.9.3 Exemplo de Funcionamento

Para mostrar o funcionamento correto do algoritmo que implementamos,apresentamos abaixo dois exemplos de grafos e o resultado obtido executando o algoritmo sobre esses dois grafos.

O primeiro deles é o próprio grafo dado de exemplo na questão, mostrado na Figura 13. Para esse grafo, obtivemos a seguinte resposta do algoritmo:

```
O No maximo de vertices independentes é 3.
São eles: 2 3 4
```

O segundo deles é uma versão do grafo dado de exemplo na questão só que completamente conectado, mostrado na Figura 14. Para esse grafo, obtivemos a seguinte resposta do algoritmo:

```
O No maximo de vertices independentes é 1.
São eles: 0
```

2.9.4 Avaliação das Soluções

Nessa seção apresentamos uma avaliação da qualidade das resposta obtidas escutando nosso algoritmo de remoção de cliques para o problema do conjunto máximo de vértices independentes. Para esse algoritmo foi mostrada anteriormente, na seção 2.9.2, a aproximação analítica apresentada por [8, 25] que fala que a qualidade das soluções é garantidamente de $O(\frac{n}{(\log n)^2})$.

De qualquer forma, apresentamos a qualidade das respostas do algoritmo de remoção de clique medidas empiricamente. Para isso, foram gerados grafos aleatórios com grau de conectividade variando entre 1 e 3 e a quantidade de vértices variando entre 10 e 70. Para cada um desses grafos foram feitas 5 execuções e a solução final foi feita com base na média das soluções encontradas em cada uma das execuções.

O primeiro desses resultados pode ser observado no gráfico na Figura 26 onde apresentamos a qualidade das soluções para os grafos de conectividade 1. Para as medições realizadas temos que a qualidade da respostas obtidas pela solução do algoritmo nunca foi inferior a 65%, não contradizendo a aproximação mostrada anteriormente. No entanto mostra que o mesmo possui uma qualidade não muito boa para alguns casos se comparada com as soluções encontradas pelo algoritmo guloso grafos com conectividade 1.

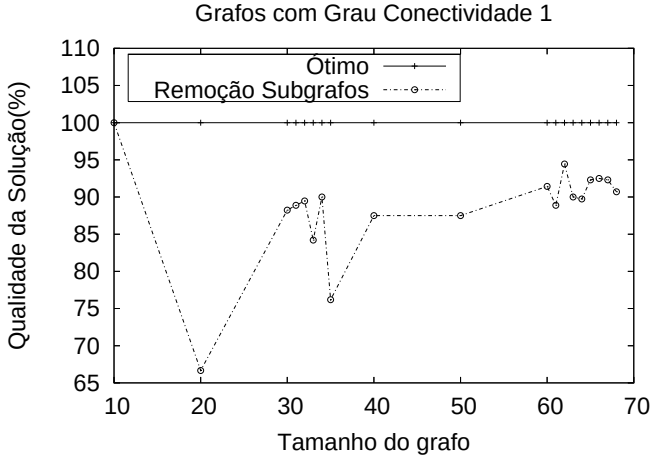

Figura 26: Qualidade da Solução - Remoção Cliques

Outros resultados podem ser observados no gráfico na Figura 27 onde apresentamos a qualidade das soluções para os grafos de conectividade 2. Para as medições realizadas temos que a qualidade da respostas obtidas pela solução do algoritmo nunca foi inferior a 65%, não contradizendo novamente a aproximação mostrada em

seções anteriores. No entanto, mais uma vez temos o mesmo possui uma qualidade não muito boa para alguns casos se comparada com as soluções encontradas pelo algoritmo guloso grafos com conectividade 1.

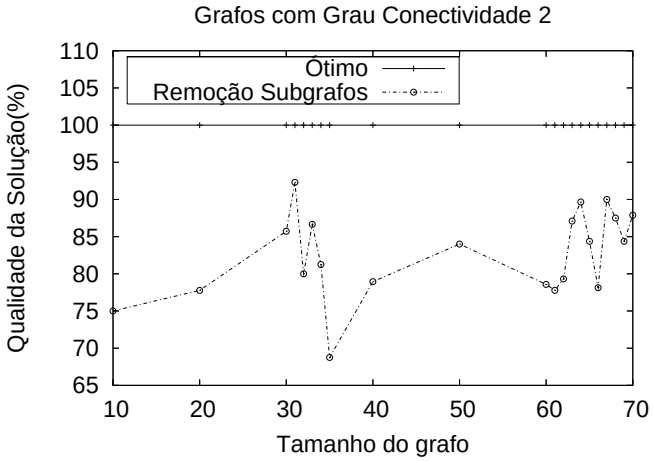

Figura 27: Qualidade da Solução - Remoção Cliques

Outros resultados podem ser observados no gráfico na Figura 28 onde apresentamos a qualidade das soluções para os grafos de conectividade 3. Para as medições realizadas temos que a qualidade da respostas obtidas pela solução do algoritmo nunca foi inferior a 65%, não contradizendo novamente a aproximação mostrada em seções anteriores. No entanto, mais uma vez temos o mesmo possui uma qualidade não muito boa para alguns casos se comparada com as soluções encontradas pelo algoritmo guloso grafos com conectividade 1 e 2.

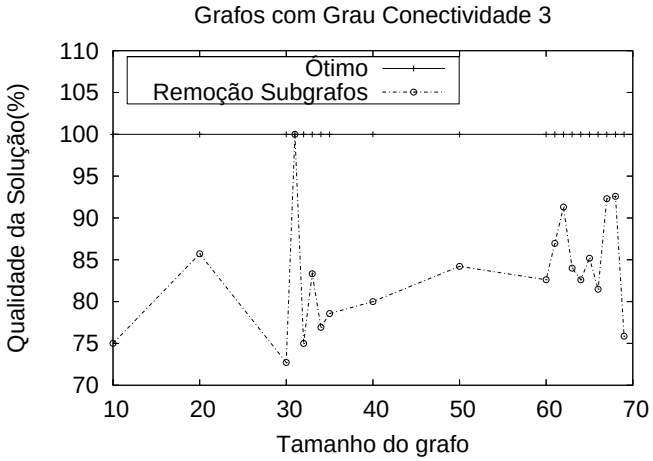

Figura 28: Qualidade da Solução - Remoção Cliques

Por fim, fizemos uma análise de eficiência da aproximação apresentada. Para isso, foram gerados grafos aleatórios com grau de conectividade variando entre 1 e 3 e a quantidade de vértices variando entre 10 e 70. Para cada um desses grafos foram feitas 1 execução com o algoritmo que obtém a solução ótima e 5 execuções

com o algortimo de remoção de clique e a solução final desse algoritmo foi obtida com base na média das soluções encontradas em cada uma das execuções. Em posse desses dados, ordenamos as soluções ótimas encontradas em oredem crescente e para cada uma delas plotamos um ponto no gráfico referente ao razão da solução ótima pela solução encontrada, uma vez que o problema do máximo conjunto de vértices independentes é um problema de maximização:

$$R_A = \frac{S^*(I)}{S(I)}$$

Mostramos na seção 2.9.2, a aproximação analítica apresentada por [8, 25] que fala que o fator de aproximação da solução para o algoritmo de remoção de cliques de da ordem de $O(\frac{n}{(\log n)^2})$, o que significa que, como o problema é de maximização, a equação $R_A = \frac{S^*(I)}{S(I)}$ nunca será maior que $O(\frac{n}{(\log n)^2})$. Pela definição da notação O, temos que, existe uma constante b tal que multiplicada por $\frac{n}{(\log n)^2}$, faz com que a mesma domine assintoticamente todas as razões $R_A = \frac{S^*(I)}{S(I)}$. Assim plotamos a curva para $\frac{n}{(\log n)^2}$, com log na base e, e ajustamos a mesma através de sua multiplicação por uma constante até que todos os pontos da razão plotados estivessem abaixo dessa curva. Para os nosso resultados encontramos $b = 0.77$, ou seja, para os nossos dados medidos, a função deve multiplicada por 0.77. No entanto não podemos afirmar que essa constante seja suficiente sempre, mas com esse resultado podemos que afirmar que a constante deve ser pelo menos 0.77. Quanto mais experimentos são executados, mais pontos são encontrados e maior pode ser essa constante, mas novamente frisamos que para nossos experimentos essa constante foi suficiente o que garante que pelo menos a constante deve ser 0.77 para log na base e, mas nada impede que esse valor seja maior ou menor.

No gráfico da Figura 29 temos que a cruva com legenda Limite Superior representa a função $f(x) = 0.77(\frac{n}{(\log n)^2})$ e a curva com legenda Limite Inferior representa a função $f(x) = 1$. Cada ponto representa um razão $R_A = \frac{S^*(I)}{S(I)}$. Dessa forma, nenhum ponto pode ser inferior a curva de Limite inferior, caso contrário teríamos um caso absurdo onde a solução encontrada pelo algoritmo de remoção de sub-cliques é melhor que a solução ótima. Ao mesmo tempo, nenhum ponto pode estar acima da curva de limite superior, caso contrário a curva não representaria o fator de aproximação.

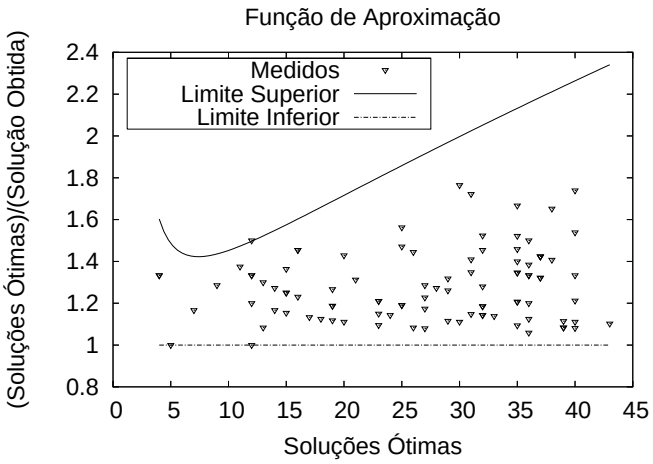

Figura 29: Análise de Eficiência

O que podemos concluir é que a qualidade das soluções geradas pelo algoritmo de remoção de cliques não se mostrou tão boa para os testes realizados, no então tal algoritmo tem uma garantia de qualidade mínima, o que pode ser muito importante a medida em que aumentamos o tamanhos dos grafos e a conectividade dos mesmos.

2.10 Algoritmos Genéticos

2.10.1 Introdução

Algoritmos Genéticos são procedimentos de procura em paralelo inspirado no mecanismo de evolução natural de sistemas [16, 43, 20]. Ao contrário das mais tradicionais técnicas de otimização, eles trabalham com uma população de pontos, que na terminologia de algoritmos genéticos, são chamados de cromossomos ou indivíduos. Na implementações mais simples e mais populares, cromossomos são uma longa cadeia de bits. Cada indivíduo tem uma "saúde"associado ao mesmo, valor que determina a probabilidade do mesmo sobreviver para a próxima geração: aqueles com melhor "saúde", possuem uma probabilidade maior de sobreviver. Algoritmos genéticos começam com uma população inicial de membros, geralmente escolhidos randomicamente, e que evoluem através de operações como reprodução e mutação. A reprodução consiste em escolher dois membros aleatórios da população para gerar um novo individuo para a próxima geração. A combinação dos dois pode ser feita de várias formas, a mais comum é o Cross-Over, no qual parte da de cada um dos dois indivíduos selecionados são combinados para forma um novo indivíduo. Finalmente a operação de mutação é aplicada que se baseia na inversão de alguns valores do indivíduo de forma randômica. Assim, a programação evolutiva é uma meta-heurística que se fundamenta em uma analogia com processos naturais de evolução, nos quais, dada uma população, os indivíduos com características genéticas melhores têm maiores chances de sobrevivência e de produzirem filhos cada vez mais aptos, enquanto indivíduos menos aptos tendem a desaparecer. Cada indivíduo novo é submetido a uma mutação. Dessa forma, os indivíduos da população se tornam mais aptos e as recombinações genéticas tendem a gerar indivíduos melhores.

2.10.2 Trabalhos Relacionados

Uma das tentativas de resolver o problema do máximo clique utilizando algoritmos genéticos foi feita em 1993 por Carter and Park [13] que depois de apresentar a ineficiência de uma algoritmo genético simples [20] para resolver o problema de encontrar grandes cliques, mesmo em pequenos grafos, eles introduzem algumas modificações na tentativa de melhorar a eficiência do mesmo. No entanto,mesmo com esse esforço, eles] não conseguiram resultados satisfatórios e concluíram que algoritmos genéticos necessitam de algumas alterações mais pesadas para se tornarem competitivos com as propostas tradicionais, além de serem um tanto quanto caros computacionalmente falando.

Mais tarde, no estudo em [36] os autores chegam a conclusão que algoritmos genéticos são menos eficientes que *simulated annealing*, no entanto, nessa mesma época, Back e Khnuri [1] trabalharam com o problema do máximo conjunto de vértices independentes e chegaram na conclusão oposta. Utilizando um proposta geral e direta de algoritmos genéticos chamado de GENEsYs e com uma função de "saúde"mais elaborada, que inclui penalidades gradativas para penalizar soluções inválidas, eles conseguiram resultados interessantes sobre grafos randômicos e regulares com até 200 vértices. Esses resultados indicaram que a escolha da função que determina a "saúde"dos indivíduos é crucial para prover algoritmos com resultados satisfatórios.

Murph et al. [37] também realizaram experimentos com algoritmos genéticos um novo modelo baseado em *partial copy crossover* e assim modificaram o operador de mutação. Entretanto, na apresentação dos resultados foi feita em grafos pequenos(no máximo 50 vértices), o que tornou difícil avaliar efetivamente o algoritmo. Bui e Eppley [11] obtiveram resultados bastante satisfatórios utilizando uma estratégia híbrida a qual incorpora uma otimização local a cada nova geração do algoritmo genético e uma fase de preprocessamento de ordenação de vértices. Eles testaram essa proposta e alguns grafos da DIMACS obtendo bons resultados comparados com [30].

Ao invés de usarem uma representação binária padrão dos cromossomos, Foster e Soule [42] utilizaram um esquema baseado em inteiros. Além disso, eles utilizaram uma função mais elaborada de "saúde"similar a utilizada por Carter and Park [13]. Os resultados foram interessantes, no entanto não foram comparados com as heurísticas mais tradicionais do problema. Fleurent e Ferland [19] um sistema de propósito mais geral para resolver problemas como o de coloração de grafos, máximo clique e satisfabilidade. A medida que o problema do máximo clique se tornava mais interessante, eles realizavam experimentos utilizando um esquema híbrido de procura, incorporando *tabu search* e outras técnicas de de busca local, assim como técnicas alternativas de mutação. Os resultados apresentados são realmente muito bons, mas o tempo de execução é um pouco alto.

Em [24], Hifi modificam um algoritmo genético básico em vários aspectos: um operador de crossover que gera dois filhos diferentes; um operação de mutação que é substituída por um uma heurística de transição de praticabilidade que dificulta a estabilização da população. Resultados experimentais em grafos randômicos e também em alguns grafos do DIMACS são reportados e validam a proposta.

Finalmente, Marchiori [33] apresenta uma heurística simples que consiste na combinação de algoritmos genéticos com um procedimento guloso. Neste trabalho

existe uma divisão do trabalho, a procura por grandes sub-grafos e a procura por um clique. Essa proposta foi avaliada sobre diversos grafos do DIMAC com bons resultados, tanto em questão de desempenho quanto na qualidade das soluções.

O que podemos ver por meio dos trabalhos relatados acima é que ainda não existe um consenso com relação ao uso de algoritmos genéticos para resolver o problema do máximo clique e do máximo conjunto de vértices independentes. Existem alguns trabalhos que obtiveram bons resultados, acima como outros que os resultados não foram tão satisfatórios. Existe ainda uma lacuna a ser preenchida com relação a aplicação de algoritmos genéticos nesse tipo de problema. No entanto, algumas conclusões já existem como a de que a operação de mutação é crucial para se obter bons resultados, além de que a combinação dessas técnicas com outras tradicionais podem resultar em boas soluções. Dessa forma, apresentamos a seguir nossa proposta de um algoritmo genético que utiliza alguns aspectos que ainda nos trabalhos acima relacionados não foram avaliados, como nossa função de mutação que trabalha sobre cada gene do indivíduo e nossa operação de recombinação de indivíduos que mescla várias técnicas, desde a mais ousadas como o uso de máscaras, até as mais tradicionais como o operado lógico AND sobre os indivíduos. Além disso, simplificamos nossa função de "saúde", na qual apenas indivíduos válidos são gerados e qualidade dos mesmos é medida pela quantidade de vértices que o mesmo possui.

2.10.3 Implementação

Nesta seção apresentamos o algoritmo que implementamos que se baseia na aplicação de programação evolutiva através de algoritmos genéticos [23, 2]. Para a aplicação de Algoritmos Genéticos é necessário transformar cada solução em uma seqüência de valores, sendo que cada elemento dessa seqüência representa um cromossomo de um indivíduo. Neste trabalho é proposta uma representação da solução por um vetor com N posições, onde N é o número de vértices do grafo. Cada posição do vetor representa se o vértice correspondente ao índice daquela posição está presente ou não naquela solução, ou seja, um vetor binário, de maneira que se o vértice pertence a aquela solução então atribuímos 1 para aquele cromossomo do indivíduo e 0 se não pertence. Um exemplo de um indivíduo pode ser visto na Figura 30. O indivíduo, ou seja, a solução representada por esse vetor contém os vértices 2, 3 e 4.

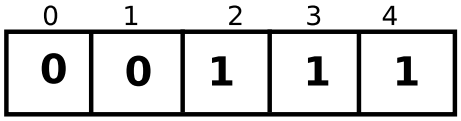

Figura 30: Exemplo de um Indivíduo(Solução)

Primeiramente o algoritmo gera uma população inicial de indivíduos onde a quantidade de indivíduos é especificada. Em nosso programa utilizamos uma definição de programa(*#define NUM_INDIVIDUOS 50*), ou seja, nesse caso nossa população possui 50 indivíduos. Em nossa proposta, apenas indivíduos válidos são gerados, ou seja, todo individuo que possui alguma aresta entre os vértices é descartado.

Assim como na evolução genética, os indivíduos de uma população passam por evoluções ao longo das gerações. Em um algoritmo genético, a quantidade de ge-

rações deve ser especificada e em nossa implementação essa especificação foi feita por uma definição de programa(*#define NUM_ GERACOES 300*). A cada geração, um fração da população é trocada, ou seja, a cada geração alguns indivíduos m̈orrem"outros ïascem", no entanto a quantidade de indivíduos a cada geração é mantida. Essa fração de troca a cada geração também é uma definição de programa(*#define FRACAO_ TROCA 2*). Em nosso algoritmo, essa troca de indivíduos a cada geração é feita por meio de dois *loops* aninhados, o mais externo é o de gerações e o mais interno o de indivíduos. Enquanto a fração de indivíduos não é completamente trocada, o programa não começa uma nova geração.

Cada troca de indivíduo em uma geração é feita da seguinte forma: seleciona-se dois indivíduos aleatoriamente da população atual e através de uma recombinação dos dois é gerado um novo indivíduo. Para cada novo indivíduo gerado, verifica-se se o mesmo é válido ou não, ou seja, dada a matriz de adjacência dos vértices do grafo, verifica-se se existe alguma aresta entre eles ou não. Caso exista, o indivíduo não é válido, e uma nova combinação deve ser feita. Isso garante que os indivíduos sempre correspondam a um solução válida para o problema.

Para esse indivíduo que surge, existe uma probabilidade do mesmo sofrer uma mutação, em nosso caso, cada cromossomo do indivíduo possui uma probabilidade de ser trocado. Feito isso, selecionamos o pior indivíduo da população atual e se o mesmo for pior que o indivíduo que acabou de ser gerado, troca-se os dois e o indivíduo que acabou de ser gerado passa a fazer parte da população. Em nosso problema, a qualidade de um indivíduo é medida pela quantidade de vértices que o mesmo possui, uma vez que queremos encontrar o maior conjunto de vértices independentes. Ao longo das gerações, temos pela teoria da evolução genética, indivíduos cada vez melhores, assim, basta ao final selecionar aquele que foi o melhor. Abaixo apresentamos um pseudo-código para o algoritmo implementado.

```
void algoritmo_genetico()
    GerarPopulacaoInicial(P);
    enquanto(i<NGeracoes) faça
        enquanto(j<NIndividuos/Fracao) faça
            pai = SelecionarPaiAleatoriamente(P);
            mae = SelecionarPaiAleatoriamente(P);
            filho = RecombinacaoAleatoria(pai,mae);
            se(probabilidade de mutação aceita) então
                mutação(filho);
            fim se
            Pior = SelecionaPiorIndividuo(p);
            se(Filho for melhor ou igual ao Pior) então
                TrocaFilhoporPior(Filho,Pior);
                j++;
            fim se
        fim-enquanto
        i++;
    fim-enquanto
    SolucaoFinal = SelecionaMelhorIndividuo(P);
fim algoritmo_genetico;
```

Para um melhor entendimento do funcionamento do algoritmo, segue abaixo uma descrição mais detalhada das funções principais do algoritmo genético implementado:

- **Gera População Inicial**: Essa é a função que gera a população inicial do algoritmo genético. Primeiramente é feita uma ordenação, utilizando o Heapsort, de todos os vértices pela quantidade de arestas que cada um deles possui. Cria-se um indivíduo que não contém nenhuma vértice. Verifica-se então se ao inserir o vértice menos conectado nesse indivíduo(solução) o mesmo é ou não uma solução viável para problema. Caso a mesma seja viável, randomicamente decide-se se esse vértice fará parte ou não desse indivíduo. Caso não seja viável, o mesmo não é inserido no indivíduo. Então seleciona-se o próximo vértice menos conectado e verifica-se então se ao inserir-lo no indivíduo em questão é ou não uma solução viável para problema. Mais uma vez, se for viável randomicamente decide-se se esse vértice fará parte ou não desse indivíduo. Caso não seja viável, o mesmo não é inserido no indivíduo. Isso é feito até que todos os vértices ordenados tenham sido avaliados para aquele indivíduo, ou seja, se todos os cromossomos foram avaliados. Ao final dessa verificação, temos um indivíduo válido. Todo esse processo é repetido na criação de cada indivíduo da população até que toda a população inicial tenha sido criada.

- **Seleciona Pai Aleatoriamente**: Essa função retorna um indivíduo da população de forma randômica.

- **Recombinação Aleatória de Indivíduos**: Essa função recombina dois indivíduos quaisquer e gera um terceiro. Essa recombinação pode ser feita de diversas formas diferentes:

    1. Por meio de uma Mascara: Para cada cromossomo do filho(indivíduo a ser gerado), ou seja, para cada posição do vetor que representa se o vértice está ou não presente no indivíduo(solução), gera-se um número aleatório entre 0 e 1. Se esse número for menor que 0.5 esse cromossomo do filho será o mesmo da mãe dele, ou seja, se a mãe tiver aquele vértice ele também o terá, se ela não o tiver, o filho também não o terá. Se esse valor randômico for maior que 0.5 esse cromossomo do filho será o mesmo do pai dele, ou seja, se o pai tiver aquele vértice ele também irá ter, se ele não o tiver, o filho também não o terá. Na Figura 31 temos uma ilustração desse método de recombinação.

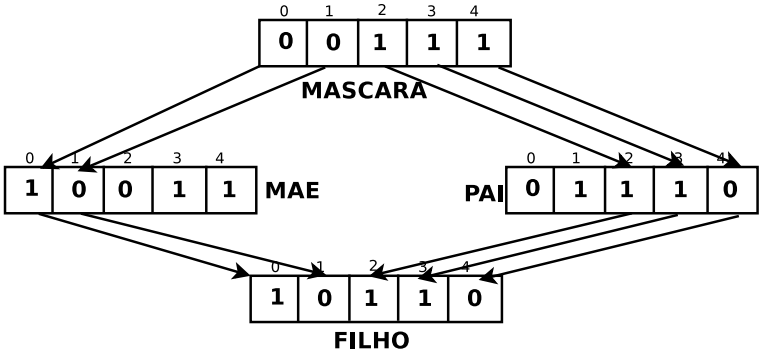

Figura 31: Recombinação por Máscara

2. Cross-Over: O filho é formado pela primeira metade dos cromossomos do pai e pela segunda metade dos cromossomos da mãe. Na Figura 32 temos uma ilustração desse método de recombinação.

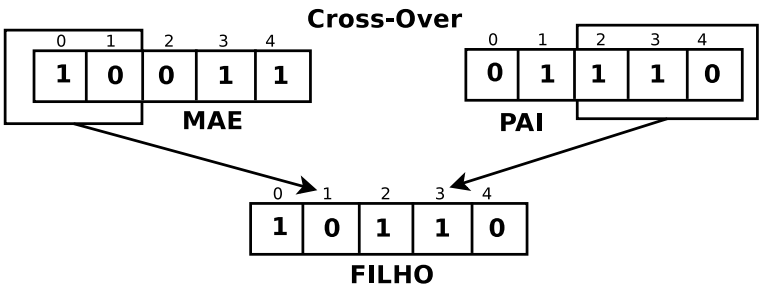

Figura 32: Recombinação por Cross-Over

3. OR: O filho é formado pela operação lógica OR entre os cromossomos do pai com os da mãe Na Figura 33 temos uma ilustração desse método de recombinação.

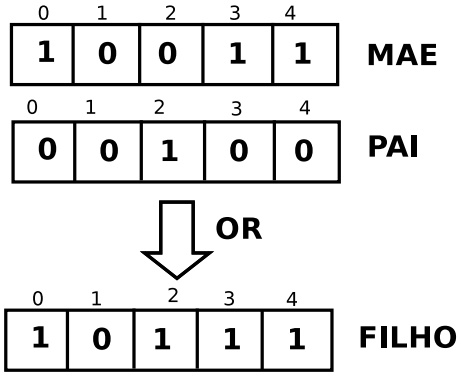

Figura 33: Recombinação por OR

4. XOR: O filho é formado pela operação lógica XOR entre os cromossomos do pai com os da mãe Na Figura 34 temos uma ilustração desse método de recombinação.

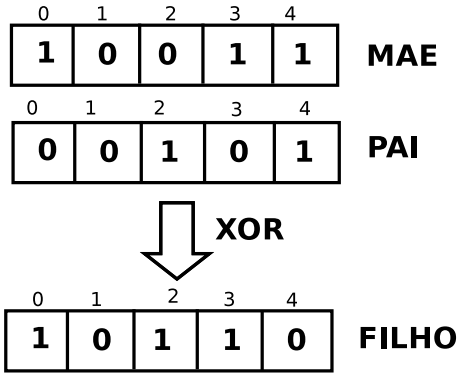

Figura 34: Recombinação por XOR

5. AND: O filho é formado pela operação lógica AND entre os cromossomos do pai com os da mãe Na Figura 35 temos uma ilustração desse método de recombinação.

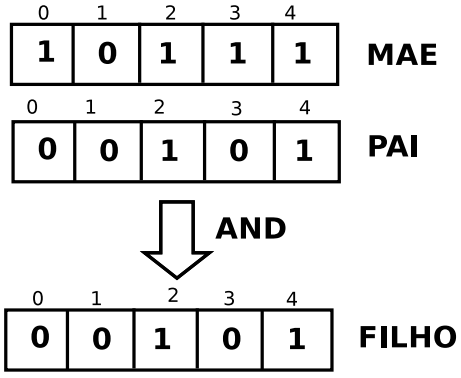

Figura 35: Recombinação por AND

A recombinação é tentada nessa ordem, se através da máscara gerar um indivíduo inválido, tenta-se então o Cross-Over, se novamente falhar, tenta-se o OR e depois o XOR e por ultimo o AND. Se, mesmo depois de todas essas tentativas não surgir um indivíduo válido, um novo pai e uma nova mãe devem ser selecionados.

- **Mutação**: Está é a função que realiza uma mutação no indivíduo. Para cada cromossomo do indivíduo, existe uma probabilidade de 10% daquele cromossomo ser alterado, ou seja, se o vértice pertence ele deixa de pertencer, se não existe passa a pertencer. Se a alteração gera um indivíduo inválido, desfaz-se a mutação. Essa probabilidade é verificada para cada cromossomo do individuo.
- **Seleciona Pior Indivíduo**: Nesse procedimento primeiramente é feita uma ordenação entre os indivíduos que fazem parte da população pela quantidade

de vértices que os mesmos possuem, ou seja, daqueles com menos vértices para aqueles com mais vértices. Mais uma vez, utilizamos o Heapsort. Aqueles indivíduos com menos vértices são tidos como indivíduos(soluções) ruim. Dessa forma, utilizando uma função randômica que escolhe aleatoriamente um dos piores indivíduos e retorna o mesmo. A opção por fazer essa escolha aleatória entre os piores e não retornar sempre o pior dos piores foi feita para manter o algoritmo o mais parecido com a evolução genética, onde nem sempre os piores sempre morrem. Com isso não perdemos as características desses indivíduos e não perdemos conseqüentemente possíveis soluções melhores que esses indivíduos possam resultar.

- **Compara dois Indivíduos**: Dois indivíduos(o filho e o pior indivíduo) são passados como entrada para essa função e a mesma compara qual dos dois indivíduos é o melhor. Essa comparação é feita pela quantidade de vértices que cada um deles possui. Se o filho é o melhor a função retorna 1, caso contrário retorna 0.

- **Troca Filho com o Pior**: Para essa função, dada dois indivíduos, o filho e o pior indivíduo, ela remove o pior indivíduo da população e insere o filho a mesma.

- **Seleciona Melhor Indivíduo**: Nesse procedimento primeiramente é feita uma ordenação entre os indivíduos que fazem parte da população pela quantidade de vértices que os mesmos possuem, ou seja, daqueles com menos vértices para aqueles com mais vértices. Mais uma vez, utilizamos o Heapsort. Ao final dessa ordenação, basta retornar aquele indivíduo da população que contém o maior número de vértices.

- **Verifica Indivíduo é Válido**: Essa função recebe como parâmetro o indivíduo e a matriz de adjacência. Para cada par de vértices que pertencem ao indivíduo, verifica-se se existe uma aresta entre esses dois indivíduos. Encontrada a primeira aresta, o procedimento retorna imediatamente que o indivíduo é inválido, retorna 0, e a aborta o restante da verificação. Caso não encontre nenhuma aresta, retorna que o indivíduo é válido, retorna 1.

Em nossa implementação colocamos ainda um algoritmo guloso como uma última tentativa de melhorar a solução final. Esse algoritmo guloso é bastante parecido com o já descrito acima, referente a solução gulosa. Ao final da execução do algoritmo genético, ordenamos todas os vértices do grafo pela quantidade de arestas. Do menos conectado ao mais conectado, tentamos inserir o vértice na solução, caso o mesmo já não esteja na solução. Se ao inserir tal vértice na solução, a mesma se mantiver correta, o vértice passa a fazer parte da solução. Caso contrário, o mesmo não é inserido. Isso é feito para todo vértice do grafo, e ao final isso pode levar a uma boa solução, caso a população do algoritmo não tenha se estabilizado.

2.10.4 Avaliação do Algoritmo

Existem vários parâmetros do algoritmo genético que podem ser escolhidos para melhorar o seu desempenho, adaptando-o às características particulares de determinadas classes de problemas. Entre eles os mais importantes são: o tamanho da

população, o número de gerações, a probabilidade da escolha da recombinação e a probabilidade de mutação. A influência de cada parâmetro no desempenho do algoritmo depende da classe de problemas que se está tratando. Assim, a determinação de um conjunto de valores otimizado para estes parâmetros dependerá da realização de um grande número de experimentos e testes. Na maioria da literatura os valores encontrados estão na faixa de 60 a 65% para a probabilidade de cross-over e entre 0,1 e 5% para a probabilidade de mutação. O tamanho da população e o número de gerações dependem da complexidade do problema de otimização e devem ser determinados experimentalmente. No entanto, deve ser observado que o tamanho da população e o número de gerações definem diretamente o tamanho do espaço de busca a ser coberto, o que aumenta assim o tempo de execução desses algoritmos. Existem estudos que utilizam um AG como método de otimização para a escolha dos parâmetros de outro AG, devido à importância da escolha correta destes parâmetros.

Observando a pseudo-código código apresentado na descrição da implementação, podemos ver que existe uma dependência muito grande do nosso algoritmo de funções randômicas. A maioria das ações são tomadas com base em probabilidade igualmente distribuídas, desta forma, torna-se extremamente complicado, ou até mesmo impossível, calcular a complexidade desse tipo de algoritmo. São essas funções de probabilidade que irão determinar, juntamente com espaço de busca a ser coberto(tamanho da população e número de gerações), a complexidade de tempo desses algoritmos. Não existe um pior caso, nem um melhor caso para essa classe de algoritmos, tudo depende das decisões tomadas randomicamente como população inicial, mutação, seleção de indivíduos para recombinação ao longo das gerações e o quão rápido a população converge para uma população de super-indivíduos(máximo global ou um máximo local). Dessa forma, existe um compromisso entre espaço de busca a ser coberto, tempo de execução e qualidade da solução. Se o espaço de busca a ser coberto é muito grande, a probabilidade de termos uma solução de melhor qualidade é muito grande, a mesmo tempo, isso poder ser caro, computacionalmente falando. No entanto, se o espaço de busca a ser coberto for pequeno, o tempo de execução pode ser baixo, mas a qualidade da solução pode não ser muito boa.

Dessa forma realizamos alguns testes até determinar qual seria os valores que utilizaríamos para os parâmetros citados acima. Nossos testes se basearam em grafos aleatórios com grau de conectividade variando entre 1 e 3 e a quantidade de vértices variando entre 10 e 70. Para o Cross-Over, nossa implementação é um pouco diferente das tradicionais. Um novo indivíduo só surge a partir de uma recombinação de outros dois indivíduos da população, e como descrito anteriormente, são feitas até 5 tentativas de recombinação para cada par de indivíduos antes de desistirmos de recombiná-los. Um dessas tentativas é p cross-over. Assim, em nossa implementação esse não foi um parâmetro a ser calibrado.

Foram feitos inúmeros testes, e ao final, para essa classe de grafos, determinamos que os valores que resultaram em um melhor custo benefício entre tempo de execução e qualidade da solução pode ser observados na Tabela 6

Fazendo-se uma análise mais simplificada da complexidade do algoritmo implementado,temos que que a função que gera a população inicial tem uma complexidade de $O(v^2)$, onde v é o número de arestas, uma vez que a mesma varre todos os vértices do grafo para montar a população inicial. Essa é a única função que se encontra fora dos loops de geração e troca de indivíduos. Dentro desse loop temos a função

Tabela 6: Relação dos Valores dos Parâmetros

Parâmetro	Valor
Número de Gerações	300
Número de Independentes	50
Prob. de Mutação	10%por gen

que escolhe indivíduos para serem recombinados que tem complexidade $O(1)$, uma vez que a escolha é feita através de uma função randômica.

Temos também a função de recombinação, que é executada no pior dos casos 5 vezes, onde 5 é o número de funções de recombinação existentes. Cada uma dessas funções tem uma complexidade de $O(v)$, um vez que todos os vértices de cada indivíduo deve ser percorrida. Assim para esse procedimento temos uma complexidade total de $5.O(v^2)$, assim $O(v^2)$. Temos que, para cada recombinação feita, é feita também uma avaliação se o indivíduo gerado é ou não válido. Essa função de avaliação do indivíduo tem uma complexidade de $O(v^2)$, uma vez que cada vértice da solução avaliada deve ser combinado com os outros todos da solução para se ter a certeza de que não existe nenhuma aresta entre eles.

Temos ainda a função de mutação que percorre cada um dos vértices do indivíduo verificando se a probabilidade de mutação de cada um deles deve ser feita ou não. Em caso afirmativo, o indivíduo resultante da mutação deve ser avaliado se é ou não válido. Assim, temos como caso extremo, todos os vértices sendo alterados uma verificação sendo feita para cada alteração. Assim a complexidade dessa função é dada por $O(v).O(Vv^2)$, onde $O(v^2)$ e a complexidade da função de validação, assim a complexidade é $O(v^3)$.

E por fim temos as funções que seleciona o pior individuo e a função que troca os indivíduos. A primeira tem uma complexidade de $O(v.NUM\_INDIVIDUOS)$, como o NUM_INDIVIDUOS pode ter o tamanho do número de vértices, vamos fazer uma simplificação e assumir que a complexidade da mesma é de $O(v^2)$. A segunda é uma função simples, que remove um indivíduo e insere um novo, vértice a vértice, ou seja, complexidade $O(v)$.

Assim, dentro desses dois loops temos que a complexidade é dada pela função de maior complexidade, assim $MAX(O(v), O(v^2), O(v^2, O(v^3)))$ temos a complexidade $O(v^3)$. Assim temos que, essa função é executada por N troca de indivíduos por M gerações. Mais uma vez, simplificando o cálculo, temos que a complexidade é dada por $O(v^3).N.M$ é transformada em $O(v^5)$. No fim do programa, ainda temos um algoritmo guloso que tenta melhorar a solução com complexidade $O(v^2)$ e a função que escolhe o melhor indivíduo com complexidade $O(v)$. Assim temos que o complexidade do algoritmo genético implementado pode ser dada de forma bastante simplificada por $O(v^5)$.

Sobre essa complexidade, ainda temos uma observação importantíssima a ser feita. A complexidade é polinomial, no entanto o tempo de execução pode ser alto, uma vez que a troca de indivíduo entre uma geração e outra pode ser demorada se a convergência da população for lenta. Como dito anteriormente, existem diversos fatores que influenciam na qualidade da resposta e na velocidade com que a mesma

é obtida, uma vez que a maioria das decisões são tomadas com base em funções de probabilidade. Assim, o algoritmo genético, apesar de poder ser dado de forma bastante simplificada como sendo polinomial, na prática, o tempo de execução pode ser bastante alto.

Na Figura 36 apresentamos uma análise para o tempo de execução do algoritmo genético implementado para grafos de tamanhos variados com grau de conectividade 1. Como mencionado anteriormente, utilizamos os parâmetros apresentados na Tabela 6 . Para cada um dos grafos foram feitas 5 execuções diferentes e tomado o tempo médio dessas execuções. Observando esse gráfico, temos como legenda que FB refere-se aos tempos medidos para o algoritmo de força bruta, GM refere-se aos tempos coletados para o algoritmo GenMax e AG refere-se aos tempos do algoritmo genético. Enquanto os tempos de execução dos algoritmos que geram a solução ótima crescem rapidamente, temos que o tempo de execução para o algoritmo genético cresce mais lentamente, apesar de um tempo maior que as demais soluções para os problemas menores. Podemos observar também que para problemas maiores, o tempo de execução para o algoritmo genético variou bastante, o que comprova o que foi dito anteriormente sobre a influência que as escolhas probabilísticas têm sobre o desempenho do algoritmo, chegando em alguns casos a ser tão ruim quanto um algoritmo que obtém a solução ótima.

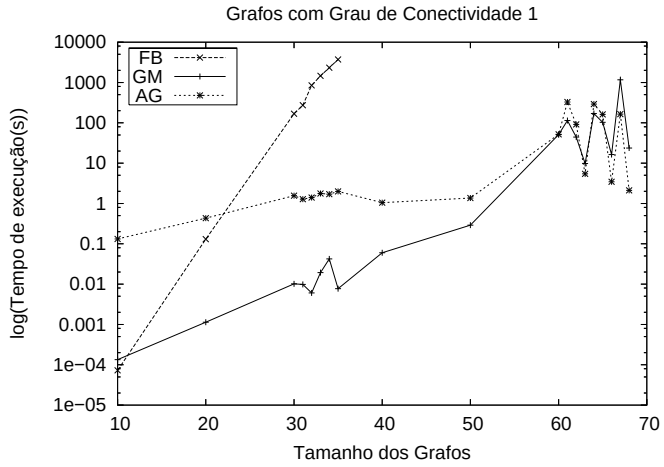

Figura 36: Tempo de Execução - Alg. Genéticos - Conectividade 1

Na Figura 37 apresentamos uma análise para o tempo de execução do algoritmo genético implementado para grafos de tamanhos variados com grau de conectividade 2. Mais uma vez utilizamos os parâmetros apresentados na Tabela 6 . Para cada um dos grafos foram feitas 5 execuções diferentes e tomado o tempo médio dessas execuções. Observando o gráfico, mais uma vez podemos perceber que o tempo de execução do algoritmo genético cresce bem mais lentamente que os demais algoritmos a medida que aumentamos o tamanho da entrada. No entanto, é importante ressaltar que a medida que cresce o tamanho do grafo, os parâmetros escolhidos podem não ser suficientes para que tenhamos uma solução boa para o problema. Assim, a medida que aumentamos o tamanho dos grafos que o algoritmo genético deve resolver, devemos também realizar vários experimentos com o mesmo para que novos parâmetros sejam escolhidos para melhorar a solução, e isso pode acarretar

em um tempo de execução maior também.

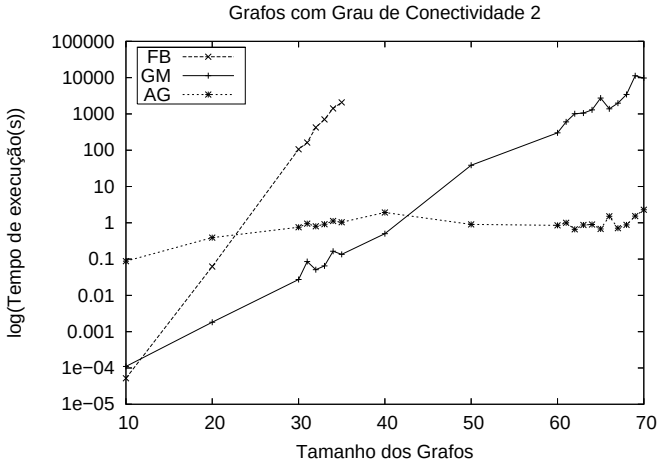

Figura 37: Tempo de Execução - Alg. Genéticos - Conectividade 2

Na Figura 38 apresentamos uma análise para o tempo de execução do algoritmo genético implementado para grafos de tamanhos variados com grau de conectividade 3. Mais uma vez utilizamos os parâmetros apresentados na Tabela 6 . Para cada um dos grafos foram feitas 5 execuções diferentes e tomado o tempo médio dessas execuções. Observando o gráfico, mais uma vez podemos perceber que o tempo de execução do algoritmo genético cresce bem mais lentamente que os demais algoritmos a medida que aumentamos o tamanho da entrada, se tornando uma excelente opção para resolver o problema. No entanto, é importante ressaltar que a medida que cresce o tamanho do grafo, os parâmetros escolhidos podem não ser suficientes para que tenhamos uma solução boa para o problema. Assim, a medida que aumentamos o tamanho dos grafos que o algoritmo genético deve resolver, devemos também realizar vários experimentos com o mesmo para que novos parâmetros sejam escolhidos para melhorar a solução, e isso pode acarretar em um tempo de execução maior também.

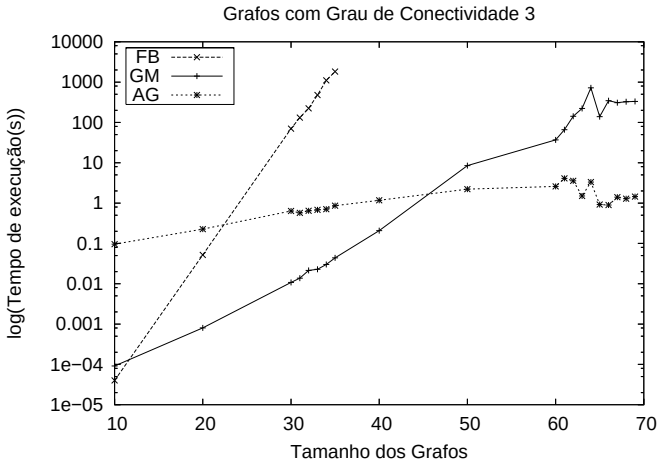

Figura 38: Tempo de Execução - Alg. Genéticos - Conectividade 3

Através dos figuras apresentadas acima, podemos perceber um crescimento bem menos no tempo de execução, mas que em alguns casos pode se tornar extremamente alto. Além disso temos que a calibragem do algoritmo, ou seja, determinar quais os valores dos parâmetros devem ser utilizados influenciam muito o tempo de execução, assim como as funções probabilísticas também influenciam. Assim, podemos concluir que na maioria dos casos, o tempo de execução do algoritmo genético é bem melhor que para os demais algoritmos, o que o torna, nesse aspecto, bastante promissor e uma boa opção.

2.10.5 Exemplo de Funcionamento

Para mostrar o funcionamento correto do algoritmo que implementamos,apresentamos abaixo dois exemplos de grafos e o resultado obtido executando o algoritmo sobre esses dois grafos.

O primeiro deles é o próprio grafo dado de exemplo na questão, mostrado na Figura 13. Para esse grafo, obtivemos a seguinte resposta do algoritmo:

```
O No maximo de vertices independentes é 3.
São eles: 2 3 4
```

O segundo deles é uma versão do grafo dado de exemplo na questão só que completamente conectado, mostrado na Figura 14. Para esse grafo, obtivemos a seguinte resposta do algoritmo:

```
O No maximo de vertices independentes é 1.
São eles: 0
```

2.10.6 Avaliação das Soluções

Nessa seção apresentamos uma avaliação da qualidade das resposta obtidas executando nosso algoritmo genético para o problema do conjunto máximo de vértices independentes. Para esse algoritmo não encontramos uma aproximação analítica que nos desce o quão próxima as soluções encontradas pelo mesmo está da solução ótima, no entanto, como apresentado na seção 2.3, sabemos que a mesma não pode ser um fator de $2^{\frac{\log\log n}{\log\log\log n}}$, caso contrário P=NP. Além disso, temos que encontrar pode uma tarefa extremamente complicada, uma vez em [7] argumenta-se que esse problema não pode ser aproximado com nada a menos que seja limitado por um n. Outro fato complicador para o cálculo da aproximação desse algoritmo está relacionada a forma com que o algoritmo trabalha condicionado a funções de probabilidade para determinar quais ações serão tomadas. Por exemplo, se optarmos por um espaço de busca a ser coberto muito pequeno, podemos chegar a um mínimo local para a solução do problema e que pode estar muito longe da solução ótima, além das mutações que são as grandes responsáveis para o algoritmo genético saia de uma solução maximal local.

Dessa forma, apresentamos a qualidade das respostas do algoritmo genético medidas empiricamente. Para isso, foram gerados grafos aleatórios com grau de conectividade variando entre 1 e 3 e a quantidade de vértices variando entre 10 e 70

e utilizando os parâmetros descritos na Tabela 6. Para cada um desses grafos foram feitas 5 execuções e a solução final foi feita com base na média das soluções encontradas em cada uma das execuções.

O primeiro desses resultados pode ser observado no gráfico na Figura 39 onde apresentamos a qualidade das soluções para os grafos de conectividade 1. Para as medições realizadas temos que a qualidade da respostas obtidas pela solução gulosa nunca foi inferior a 95%, inclusive, encontrando muitas vezes a própria solução ótima. Mesmo assim nada podemos afirmar com relação a sua aproximação, no entanto mostra que o mesmo possui uma qualidade muito boa de suas soluções para grafos com conectividade baixa. Isso mostra que a calibragem dos parâmetros do algoritmo foi bem feita, tornando o algoritmo bastante eficiente.

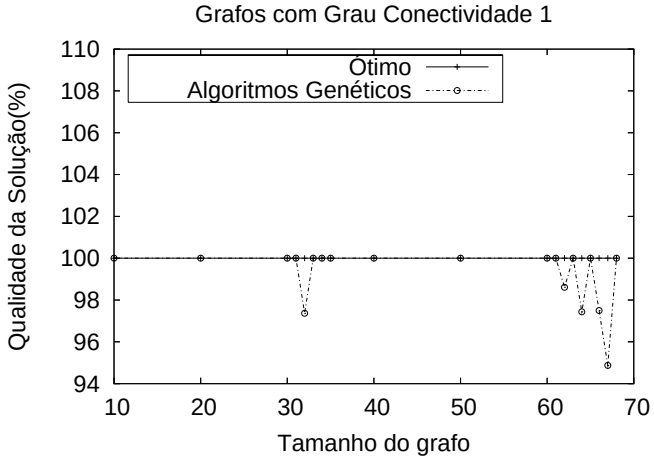

Figura 39: Qualidade da Solução - Algoritmos Genéticos

No gráfico na Figura 40 apresentamos a qualidade das soluções para os grafos de conectividade 2. Para as medições realizadas temos que a qualidade da respostas obtidas pelo algoritmo genético nunca foi inferior a 90%, inclusive, encontrando muitas vezes a própria solução ótima. Mesmo assim nada podemos afirmar com relação a sua aproximação. No entanto, se compararmos esse resultados com a qualidade das soluções encontradas para grafos de conectividade 1, percebemos uma leve queda na qualidade, de onde podemos concluir que para essa classe de grafos, a calibragem dos parâmetros do algoritmo pode ser ainda melhorada, tornando o algoritmo ainda mais eficiente.

Por fim, no gráfico na Figura 41 apresentamos a qualidade das soluções para os grafos de conectividade 3. Para as medições realizadas temos que a qualidade da respostas obtidas pelo algoritmo genético nunca foi inferior a 90%, inclusive, encontrando muitas vezes a própria solução ótima. Mesmo assim nada podemos afirmar com relação a sua aproximação. No entanto, se compararmos esse resultados com a qualidade das soluções encontradas para grafos de conectividade 1 e 2, percebemos novamente uma leve queda na qualidade, de onde podemos concluir que para essa classe de grafos, a calibragem dos parâmetros do algoritmo pode ser ainda melhorada, tornando o algoritmo ainda mais eficiente.

O que podemos concluir com relação a qualidade das soluções obtidas pelo algoritmo genético é que a mesma é muito dependente da calibragem dos parâmetros

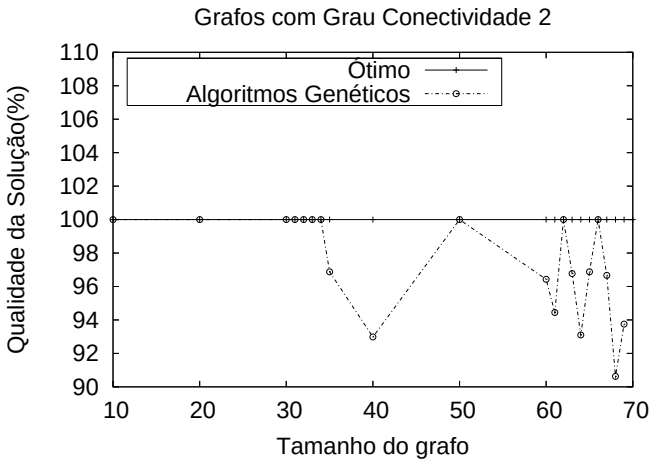

Figura 40: Qualidade da Solução - Algoritmos Genéticos

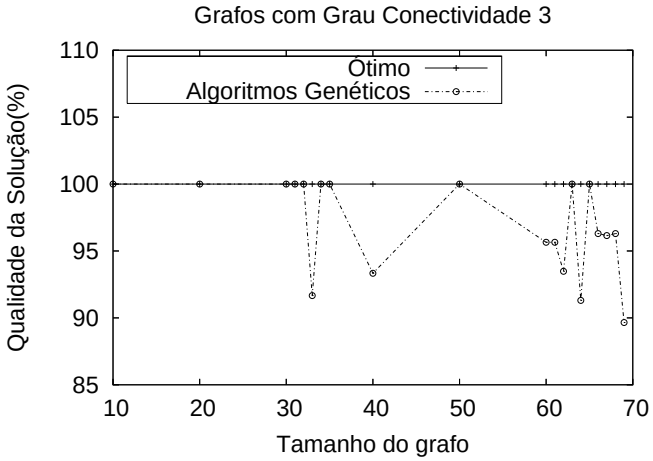

Figura 41: Qualidade da Solução - Algoritmos Genéticos

do algoritmo, no entanto essa calibragem depende das características do grafo como conectividade, tamanho etc. Para cada classe de grafos vários testes devem ser feitos para que encontremos a calibragem ideal para aquele conjunto. Na prática isso não é muito bom, pois a princípio desconhecemos algumas das características do grafo que queremos analisar. No entanto, feita essa calibragem e observando os resultados obtidos com relação a qualidade obtida e os tempos de execução, temos que algoritmos genéticos se mostram bastante promissores.

Além disso, temos que alguns pontos dos algoritmo ainda podem ser otimizados e com essas otimizações pode ser que ele se torne mais eficiente. Dentre essas alterações podemos citar a criação de outras forma de recombinação, transforma a função de seleção de pais em uma função onde indivíduos melhores possuem uma probabilidade maior de recombinação e essa probabilidade diminui a para indivíduos piores.

2.11 Comparando Todos os Algoritmos Implementados

Nesta seção apresentamos algumas comparações entre todos os algoritmos implementados. Essa comparação foi feita com base nos tempos de execução e qualidade de respostas dos mesmos. Para isso, foram gerados grafos aleatórios com grau de conectividade variando entre 1 e 3 e a quantidade de vértices variando entre 10 e 70. Para algoritmo genético mais uma vez utilizamos os parâmetros descritos na Tabela 6. Para cada um desses grafos foram feitas 5 execuções e a solução final foi feita com base na média das soluções encontradas em cada uma das execuções, tanto para os tempos quanto para a solução encontrada em cada um deles.

Primeiramente apresentamos os resultados para os grafos de conectividade 1. Na Figura 42 mostramos um gráfico que compara os tempos de execução e na Figura 43 apresentamos um gráfico que compara as qualidade das soluções encontradas para cada um dos grafos pelos algoritmos.

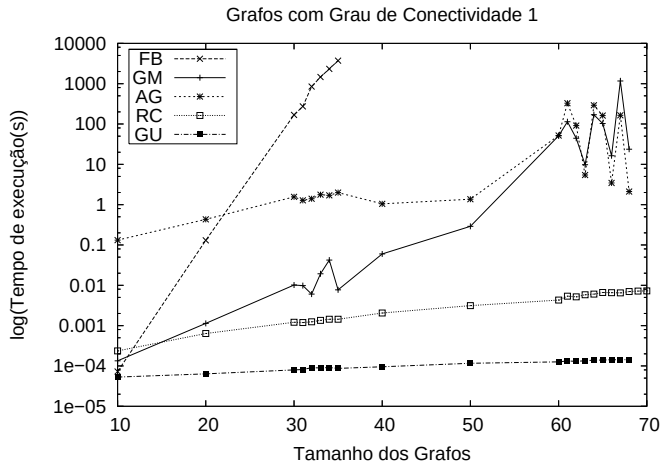

Figura 42: Tempos de Execução - Conectividade 1

Em termos da relação custo benefício entre tempo de execução e qualidade da resposta, claramente o algoritmo genético se mostra superior, mesmo que em alguns casos seu tempo de execução seja tão ruim quanto um algoritmo que gera a solução ótima. No entanto temos que o algoritmo guloso se mostra extremamente eficiente com relação ao tempo de execução, no entanto a qualidade de suas soluções não são tão boas. Já o algoritmo de remoção de cliques se mostra mais eficiente que o algoritmo genético em termos de tempo de execução, no entanto menos eficiente que o algoritmo guloso, mas em todos os casos a qualidade de suas respostas é inferior.

Nas Figura 44 e Figura 45 apresentamos os resultados para os grafos de conectividade 2 que comparam os tempos de execução e as qualidade das soluções encontradas respectivamente.

Mais uma vez, em termos da relação custo benefício entre tempo de execução e qualidade da resposta, claramente o algoritmo genético se mostra bastante superior. No entanto temos que o algoritmo guloso se mostra extremamente eficiente com relação ao tempo de execução, no entanto a qualidade de suas soluções não são tão boas. Já o algoritmo de remoção de cliques se mostra mais eficiente que o algoritmo genético em termos de tempo de execução, no entanto menos eficiente que

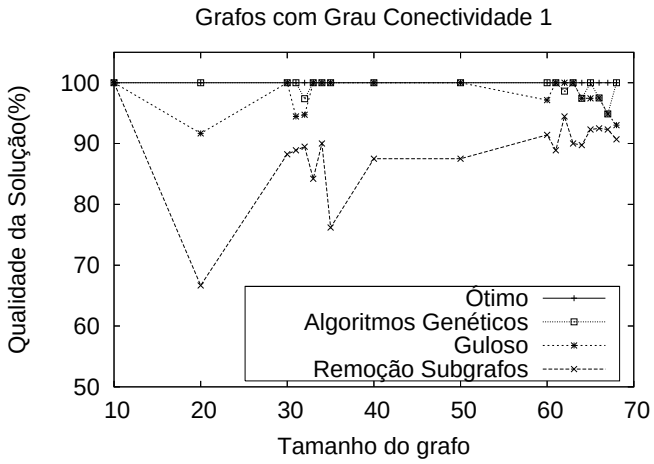

Figura 43: Qualidade da Solução - Conectividade 1

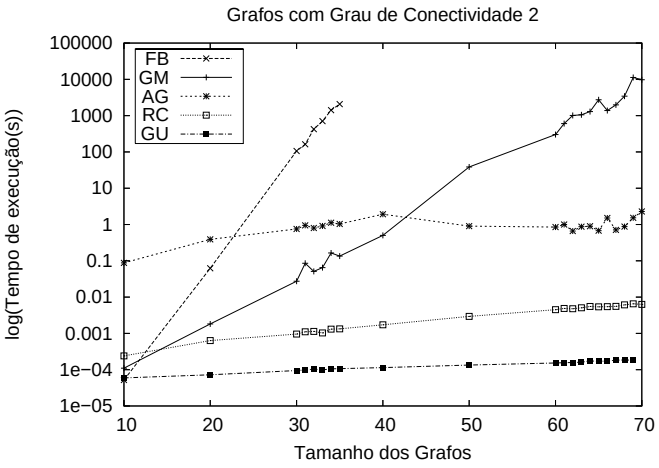

Figura 44: Tempos de Execução - Conectividade 2

o algoritmo guloso, mas em todos os casos a qualidade de suas respostas é inferior.

Mas nesse caso já podemos observar que a qualidade das soluções do guloso começa a cair com relação aos grafos de conectividade 1, assim como a qualidade das soluções do algoritmo genético, enquanto que a qualidade das soluções para o algoritmo de remoção de cliques se mantém mais ou menos constante. Isso acontece porque o algoritmo genético ainda pode ser um pouco mais calibrado para esse caso e no caso do guloso, a medida que a conectividade aumenta, com explicado anteriormente, ele tende a piorar um pouco a qualidade de suas soluções, enquanto que o algoritmo de remoção de sub-clique tem uma limite inferior para a qualidade de suas soluções, mesmo que esse limite não seja tão bom.

Nas Figura 46 e Figura 47 apresentamos os resultados para os grafos de conectividade 3 que comparam os tempos de execução e as qualidade das soluções encontradas respectivamente.

Mais uma vez, em termos da relação custo benefício entre tempo de execução e qualidade da resposta, claramente o algoritmo genético se mostra bastante superior.

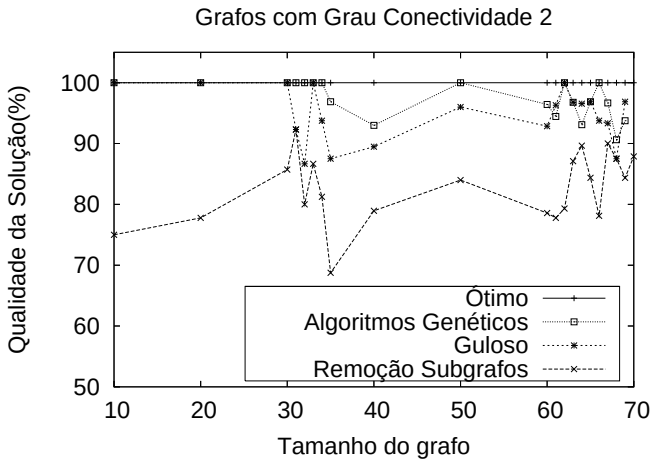

Figura 45: Qualidade da Solução - Conectividade 2

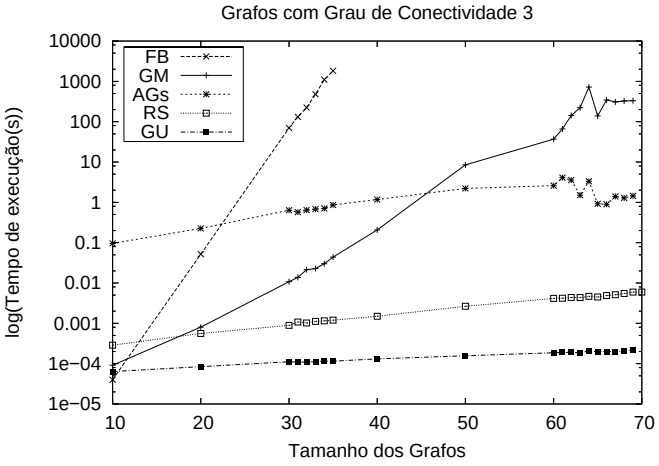

Figura 46: Tempos de Execução - Conectividade 3

No entanto temos que o algoritmo guloso se mostra extremamente eficiente com relação ao tempo de execução, no entanto a qualidade de suas soluções não são tão boas e já reduz em relação as comparações anteriores. Já o algoritmo de remoção de cliques se mostra mais eficiente que o algoritmo genético em termos de tempo de execução, no entanto menos eficiente que o algoritmo guloso, mas em todos os casos a qualidade de suas respostas é inferior.

Novamente podemos observar que a qualidade das soluções do guloso começa a cair com relação aos grafos de conectividade 1 e 2, assim como a qualidade das soluções do algoritmo genético, enquanto que a qualidade das soluções para o algoritmo de remoção de cliques se mantém mais ou menos constante. Isso acontece porque o algoritmo genético mais uma vez ainda pode ser um pouco melhor calibrado para esse caso e no caso do guloso, a medida que a conectividade aumenta, com explicado anteriormente, ele tende a piorar um pouco a qualidade de suas soluções, enquanto que o algoritmo de remoção de sub-clique tem uma limite inferior para a qualidade de suas soluções, mesmo que esse limite não seja tão bom.

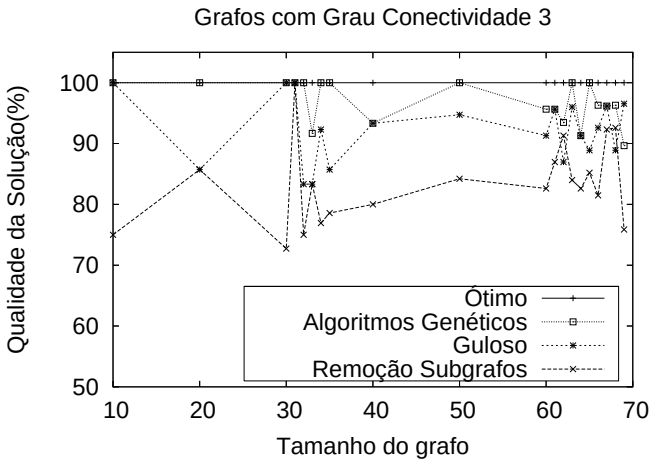

Figura 47: Qualidade da Solução - Conectividade 3

Dessas comparações podemos concluir que o algoritmo genético se mostrou bastante eficiente, tanto em relação ao tempo de execução quanto em termos de qualidade de resposta. No entanto, a cada nova classe de grafos, o mesmo deve passar por uma bateria de testes para que o ponto de equilíbrio seja encontrado. Grafos diferente podem precisar de pontos de equilíbrio muito distintos e isso, na prática pode ser bastante trabalhoso.

Podemos concluir também que apesar do algoritmo guloso apresentar soluções de qualidade superior a soluções apresentadas pelo algoritmo de remoção de cliques nos testes realizados, isso pode não ser verdade a medida em que o grau de conectividade aumenta ou até mesmo o tamanho dos grafos aumente. Isso porque o mesmo não possui qualquer garantia de qualidade, ao contrário do algoritmo de remoção de cliques.

Dessa forma, na escolha de algoritmos para resolver esse tipo de problema existe um compromisso entre qualidade de solução, tempo de execução e garantia de qualidade que deve ser levado em consideração. Se uma garantia necessária, mesmo que a mesma não seja tão boa, a opção é um algoritmo com uma aproximação como o de remoção de cliques. Se o tempo e execução é o mais importante, a opção são os algoritmos mais simples mas que não oferecem nenhuma garantia. Se o mais importante é sempre obter a melhor solução, mesmo que isso tenha um custo alto, a opção são os algoritmos que geram a solução ótima, mesmo que suas complexidade sejam não polinomiais. E por fim, se a necessidade é um casamento entre custo e qualidade e a classe do problema já bem conhecida, ou caso não seja conhecida e o trabalho de configuração inicial não seja um problema, técnicas de aproximação avançadas como *simulated annealing* e algoritmos genéticos são uma boa opção.

3 Conclusões

Neste trabalho estudamos dois problemas clássicos da computação: Distância de Palavras e Conjunto Máximo de Vértices Independentes, que é um problema NP-Completo. Para o primeiro problema implementamos uma solução baseada em programação dinâmica que consiste em resolver incrementalmente problemas menores até se atingir uma solução ótima global, nesse caso a solução utilizada foi a sugerida por Levenshtein. Para resolver esse problema foi necessário realizar um estudo mais aprofundado das técnicas de programação dinâmica.

Para o segundo trabalho, foram desenvolvidos alguns algoritmos utilizando técnicas diferentes: um que obtém-se a resposta ótima; um guloso sem garantia mínima da qualidade da solução; um de custo polinomial com garantia de qualidade mínima; e por fim um algoritmo genético. Mostramos que na escolha de qual algoritmo utilizar para resolver esse tipo de problema existe um compromisso entre qualidade de solução, tempo de execução e garantia de qualidade que deve ser levado em consideração.

Com relação ao algoritmo genético implementado, acreditamos que algumas melhoras ainda podem ser implementadas tornando ainda mais eficiente, como a criação de outras forma de recombinação, transforma a função de seleção de pais em uma função onde indivíduos melhores possuem uma probabilidade maior de recombinação e essa probabilidade diminui a para indivíduos piores. A maior parte de tempo desse trabalho foi gasto na implementação desse algoritmo e na calibragem do mesmo para a execução dos testes realizados. Acreditamos que com mais alguns ajustes no mesmo, eventualmente esse trabalho venha a se tornar o seminário da disciplina, ou até mesmo um artigo.

4 Agradecimentos

Agradeço ao grupo de estudos composto pelos alunos Alex Borges, Elisa Tuler, Guilherme Trielli, Leonardo Chaves, Marcelo Maia, Renata Braga, Bruno Coutinho, entre outros pela ajuda e discussões sobre os resultados e decisões de implementação que contribuíram positivamente para um melhor entendimento de todo trabalho.

Referências

[1] T. ack and S. Khuri. An evolutionary heuristic for the maximum independent set problem, 1994.

[2] Peter J. Angeline. Evolution revolution: An introduction to the special track on genetic and evolutionary programming. *IEEE Expert*, 10(3):6–10, 1995.

[3] A. Apostolico. String editing and longest common subsequences. HANDBOOK OF FORMAL LANGUAGES, 1997.

[4] Raymond A. Archuleta and Henry D. Shapiro. A fast probabilistic algorithm for four-coloring large planar graphs. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 595–600, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[5] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: a new characterization of np. *J. ACM*, 45(1):70–122, 1998.

[6] Egon Balas and William Niehaus. Optimized crossover-based genetic algorithms for the maximum cardinality and maximum weight clique problems. *Journal of Heuristics*, 4(2):107–122, 1998.

[7] Piotr Berman and Georg Schnitger. On the complexity of approximating the independent set problem. *Inf. Comput.*, 96(1):77–94, 1992.

[8] Ravi Boppana and Magnús M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. In J. R. Gilbert and R. Karlsson, editors, *SWAT 90 2nd Scandinavian Workshop on Algorithm Theory*, volume 447, pages 13–25, 1990.

[9] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979.

[10] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, September 1973.

[11] Thang Nguyen Bui and Paul H. Eppley. A hybrid genetic algorithm for the maximum clique problem. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 478–484, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[12] Stanislav Busygin, Sergiy Butenko, and Panos M. Pardalos. A heuristic for the maximum independent set problem based on optimization of a quadratic over a sphere. *J. Comb. Optim.*, 6(3):287–297, 2002.

[13] Robert Carter and Kihong Park. How good are genetic algorithms at finding large cliques: an experimental. Technical Report 1993-015, 1993.

[14] N. Christofides. *Graph Theory An Algorithm Approach.* Academic Press, 1975.

[15] Thomas H. Cormen, Charles L. Leiserson, Ronald L. Rivest, and Clifford Stein. *Algoritmos: Teoria e Prática.* Editora Campus, 2002.

[16] Ding-Zhu Du and P. M. Pardalos. *Handbook of Combinatorial Optimization.* Kluwer Academic Publishers, 1999.

[17] John Fletcher et.al. Gnuplot, 2005. http://www.gnuplot.info/index.html.

[18] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy. Approximating clique is almost NP-complete. pages 2–12, 1991.

[19] Charles Fleurent and Jacques A. Ferland. Object-oriented implementation of heuristic search methods for graph coloring, maximum clique, and satisfiability. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *dimacs*, pages 619–652, 1996.

[20] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[21] Karam Gouda and Mohammed J. Zaki. GenMax: An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery*, 11(3):223–242, 2005.

[22] Dan Gusfield. *Algorithms on Strings,Trees and Sequences.* Cambridge University Press, 1997.

[23] Randy L. Haupt and Sue Ellen Haupt. *Practical genetic algorithms.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[24] M. Hifi. A genetic algorithm-based heuristic for solving the weighted maximum independent set and some equivalent problems. *Journal of the Operational Research Society*, 48(6):612–622, 1997.

[25] S. Homer and M. Peinado. On the performance of polynomial-time clique-approximation algorithms on very large graphs, 1993.

[26] Arun Jagota and Laura A. Sanchis. Adaptive, restart, randomized greedy heuristics for maximum clique. *Journal of Heuristics*, 7(6):565–585, 2001.

[27] Paul Walton Purdom Jr. *The Analysis of Algorithms.* CBS College Publishing, 1984.

[28] David Karger, Rajeev Motwani, and Madhu Sudan. Approximate graph coloring by semidefinite programming. *J. ACM*, 45(2):246–265, 1998.

[29] Marek Kubale and Boguslaw Jackowski. A generalized implicit enumeration algorithm for graph coloring. *Commun. ACM*, 28(4):412–418, 1985.

[30] P.M.Pardalos L.E.Gibbons, D.W.Hearn. A continuous based heuristic for the maximum clique problem. *CLiques, Coloring, and Satisfiability-Second DIMACS Implementation Challenge*, pages 103–124, 1996.

[31] Vladimir Levenshtein. Edit distance, 2006. http://pt.wikipedia.org/wiki/Dist

[32] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966.

[33] Elena Marchiori. A simple heuristic based genetic algorithm for the maximum clique problem. In *Selected Areas in Cryptography*, pages 366–373, 1998.

[34] F. R. McMorris, Tandy J. Warnow, and Thomas Wimer. Triangulating vertex colored graphs. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 120–127, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[35] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[36] Kihong Park and Bob Carter. On the effectiveness of genetic search in combinatorial optimization. In *Selected Areas in Cryptography*, pages 329–336, 1995.

[37] Guturu Parthasarathy, Ammanamanchi Srinivasa Murthy, and V. U. K. Sastry. Clique finding - a genetic approach. In *International Conference on Evolutionary Computation*, pages 18–21, 1994.

[38] Vangelis T. Paschos. A survey of approximately optimal solutions to some covering and packing problems. *ACM Computing Surveys*, 29(2):171–209, June 1997.

[39] Marcello Pelillo. Heuristics for maximum clique and independent set. Technical Report CS-99-23, 1999.

[40] Samudrala R. and Moult J.1. A graph-theoretic algorithm for comparative modeling of protein structure. *Journal of Molecular Biology*, 279(1):287–302, 1998.

[41] Glenn C. Rhoads. Lexicographic order - implementation, 2005. http://paul.rutgers.edu/ rhoads/Code/sub_lex.c.

[42] Terence Soule, James A. Foster, and John Dickinson. Using genetic programming to find maximum cliques. Technical Report LAL 95-12, 1995.

[43] R. the, S. Research, G. University, o Department, o Science, H. in, n artificial, A. Arbor, T. University, and M. Press. Report of the systems analysis research group sys, 1975.

[44] Michael Townsend. *Discrete Mathematics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[45] Deepak Warrier, Wilbert E. Wilhelm, Jeffrey S. Warren, and Illya V. Hicks. A branch-and-price approach for the maximum weight independent set problem. *Netw.*, 46(4):198–209, 2005.

[46] Eric W Weisstein. Lexicographic order, 2005. http://mathworld.wolfram.com/LexicographicOrder.html.

[47] Eric W Weisstein. Ramsey theory, 2005. http://mathworld.wolfram.com/RamseyTheory.html.

[48] Nivio Ziviani. *Projeto de Algoritmos com Implementações PASCAL e C*. Pioneira Thomson Learning, 2004.

A Apêndice A - Como Executar os Programas

Em http://www.dcc.ufmg.br/~lcrocha/paa/tp2/src podemos encontrar o código fonte de todas as implementações realizadas nesse trabalho, separados de acordo com o problema e a solução dos mesmos.

Todos os programas foram construídos de forma padronizada e todos possuem um Makefile. Para compilar os programas, basta executar o seguinte comando:

```
make
```

Para executar o programa de distância de edição, após executar o Makefile, basta executar a seguinte linha de comando:

```
./compara_string.e -i <arquivo_entrada> -o <arquivo_saida>
```

Para executar os programa de cálculo do conjunto máximo de vértices independentes, basta entrar no diretório da solução que se deseja avaliar e, após executar o Makefile, basta executar a seguinte linha de comando:

```
./vertices_independentes -i <arquivo_entrada> -o <arquivo_saida>
```

Caso se queira executar esses algoritmos para resolver o problema do clique, basta executar a seguinte linha de comando:

```
./vertices_independentes -i <arquivo_entrada> -o <arquivo_saida> -c
```