Projeto e Análise de Algoritmos
2º Trabalho Prático

Mário Sérgio Ferreira Alvim Júnior
Matrícula 2006212359
msalvim@dcc.ufmg.br

Documentação disponível em
www.dcc.ufmg.br/∼msalvim/pos/disciplinas/paa/tp2/documentacao/msalvim-pa06tp2.ps

Cógidos fonte disponíveis em
www.dcc.ufmg.br/∼msalvim/pos/disciplinas/paa/tp2/fonte

25 de abril de 2006

Sumário

Lista de Figuras

Lista de Tabelas

List of Algorithms

UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
PROJETO E ANÁLISE DE ALGORITMOS

Última alteração: 2 de abril de 2006

Professor: Nivio Ziviani

Monitor: Fabiano C. Botelho

1º Trabalho Prático - 04/04/06 - 10 pontos

Data de Entrega: 24/04/06

Penalização por Atrazo: 1 ponto até 28/04/06 mais 1 ponto por dia útil a seguir

Observação: Toda a documentação deverá ser apresentada como uma página acessível via Web (apresente o link para acesso à documentação).

---

1 Distância de Edição

O algoritmo a seguir é útil para ser usado em corretores ortográficos. Sejam dadas duas cadeias de caracteres: $X = x_1, x_2, ..., x_m$ e $Y = y_1, y_2, ..., y_n$. O número k de operações de substituição, inserção e retirada de caracteres necessário para transformar X em Y é conhecido como distância de edição. Assim, a distância de edição $ed(X, Y)$ corresponde ao número k de operações necessárias para converter X em Y.

Por exemplo, se $X =$ `matranda` e $Y =$ `saturadas` então $ed(X, Y) = 4$. A seqüência de operações é: (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.

1.1 Algoritmo de Programação Dinâmica

Em [1] é apresentado um algoritmo de programação dinâmica para determinar a distância de edição entre duas strings X e Y, além das operações necessárias para converter X em Y. É esse algoritmo que será descrito aqui.

Definição Para duas strings $X[1..m]$ e $Y[1..n]$, uma transcrição T é uma outra string sobre o alfabeto I (inserção), D (deleção), M (casamento) e R (substituição)) que especifica uma série de operações que, aplicadas em seqência à string X a transformam na string Y.

Definição Para duas strings $X[1..m]$ e $Y[1..n]$, $D(i, j)$ é definido como a distância de edição entre $X[1..i]$ e $Y[1..j]$.

$D(i, j)$ denota o número mínimo de operações de edição para converter os i primeiros caracteres de X nos j primeiros caracteres de Y. Dessa forma o valor de $D(m, n)$ é a distância de edição entre X e Y, ou seja, o valor de $ed(X, Y)$ que está sendo procurado. Nesse caso o valor de $D(m, n)$ é calculado a partir do caso mais geral $D(i, j)$, onde $1 \leq i \leq m$ e $1 \leq j \leq n$. Essa é a base da técnica de programação dinâmica utilizada para resolver este problema. A

técnica possui três componentes essenciais: a *relação de recorrência*, a *computação tabular* e o *traceback* [1].

1.1.1 A relação de recorrência

A relação de recorrência estabelece um vínculo entre o valor de $D(i, j)$ $(i, j > 0)$ e valores de D com i, j menores. No caso em que i, j já são o menor possível, o valor de D é definido diretamente (caso base). Para o problema da distância de edição, o caso base é:

$$\begin{aligned} D(i, 0) &= i \\ D(0, j) &= j \end{aligned} \tag{1}$$

É fácil verificar que o caso base está correto. A condição $D(i, 0) = i$ corresponde ao fato de que a única maneira de se transformarem os i primeiros caracteres de X em 0 caracteres de Y é fazendo i deleções. Analogamente, a condição $D(0, j) = j$ corresponde ao fato de que a única maneira de se transformarem 0 caracteres de X nos j primeiros caracteres de Y é fazendo j inserções.

Para valores de $i, j > 0$ a relação de recorrência para é definida como:

$$D(i, j) = min[D(i - 1, j) + 1, D(i, j - 1) + 1, D(i - 1, j - 1) + t(i, j)] \tag{2}$$

Onde o valor de $t(i, j)$ é dado por:

$$t(i, j) = \begin{cases} 0, & se\ X(i) = Y(j) \\ 1, & se\ X(i) \neq Y(j) \end{cases} \tag{3}$$

Prova da correção da relação de recorrência:

Lema 1. *O valor de $D(i, j)$ deve ser $D(i, j-1)+1$, $D(i-1, j)+1$ ou $D(i-1, j-1)+t(i, j)$. Não há outras possibilidades.*

Prova: Considere uma transcrição que transforma $X[1..i]$ em $Y[1..j]$ usando o número mínimo de operações possíveis. O último símbolo desta transcrição deve ser I, D, R ou M. Se o último símbolo for I, então a última operação de edição foi a inserção do caractere $Y(j)$ no fim da primeira string (já transformada até o momento). Logo, os símbolos na transcrição antes de I especificam o número mínimo de operações para transformar $X[1..i]$ em $Y[1..j - 1]$ (se não especificassem, então a transformação de $X[1..i]$ em $Y[1..j]$ usaria um número de operações maior que o mínimo). Por definição, essa última transformação gasta $D(i, j - 1)$ operações de edição. Portanto, se o último símbolo na transcrição é I, então $D(i, j) = D(i, j - 1) + 1$.

Utilizando um raciocínio análogo, se o último símbolo da transcrição for D, então a última operação de edição foi a deleção de $X(i)$, e os símbolos na transcrição antes de D especificam o número mínimo de operações para transformar $X[1..i - 1]$ em $Y[1..j]$. Por definição, essa última transformação gasta $D(i - 1, j)$ operações de edição. Assim, se o último símbolo na transcrição é I, então $D(i, j) = D(i - 1, j) + 1$.

[1] O sentido de *traceback* é algo como "caminho de trás para adiante". Na falta de uma tradução conveniente para o termo em inglês, foi mantida a palavra original.

Se o último símbolo da transcrição for R, a última operação de edição substitui $X(i)$ por $Y(j)$, e os símbolos na transcrição antes de R especificam o número mínimo de operações para transformar $X[1..i-1]$ em $Y[1..j-1]$. Nesse caso $D(i,j) = D(i-1,j-1)+1$. Por fim, e de modo similar, se o último símbolo na transcrição for M, então $D(i,j) = D(i-1,j-1)$. Introduzindo a função $t(i,j)$ como definida em 3, pode-se combinar os dois últimos casos em um só: quando o último símbolo for R ou M então $D(i-1,j-1)+t(i,j)$.

Como o último símbolo da transcrição só pode assumir os valores I, D, R ou M, todas as possibilidades foram cobertas e o lema está provado. ∎

Lema 2. $D(i,j) \leq min[D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+t(i,j)]$.

Prova: A prova deste lema é similar à do anterior, mas o objetivo é diferente. A meta é provar construtivamente que existe uma seqüência de transormações que atinge cada um dos valores acima. Como os três valores são atingíveis, então o valor de $D(i,j)$ não pode ser maior que nenhum deles (já que, por definição, $D(i,j)$ é mínimo).

Primeiro: é possível transformar $X[1..i]$ em $Y[1..j]$ com exatamente $D(i,j-1)+1$ operações. Basta transformar $X[1..i]$ em $Y[1..j-1]$ com o mínimo de operações possíveis e então usar uma operação a mais para inserir o caractere $Y(j)$ no fim. Por definição, o número de operações necessárias para fazer essa transformação é $D(i,j-1)+1$. De maneira similar, é possível transformar $X[1..i]$ em $Y[1..j]$ com exatamente $D(i-1,j)+1$ operações. Para isso basta transformar $X[1..i-1]$ em $Y[1..j]$ com o mínimo de operações possíveis e então usar uma nova operação para retirar o caractere $X(i)$ no fim. Também por definição, o número de operações necessárias para fazer essa transformação é $D(i-1,j)+1$. Por fim, é possível fazer a transformação de $X[1..i]$ em $Y[1..j]$ com exatamente $D(i-1,j-1)+t(i,j)$ operações usando um argumento análogo. ∎

Os lemas 1 e 2 implicam diretamente a correção da relação de recorrência 2 para $D(i,j)$.

Teorema 1. *Para* $i,j > 0$, $D(i,j) = min[D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+t(i,j)]$.

Prova: O lema 1 diz que $D(i,j)$ deve ser *igual a um dos três valores* $D(i-1,j)+1$, $D(i,j-1)+1$ ou $D(i-1,j-1)+t(i,j)$. O lema 2 diz que $D(i,j)$ deve ser *menor ou igual ao menor destes valores.* Portanto segue diretamente que $D(i,j)$ deve ser *igual ao menor dos valores.* ∎

Isso completa a discussão sobre o primeiro componente do método de programação dinâmcia para o problema, a relação de recorrência.

1.1.2 A computação tabular

O segundo componente do método de programação dinâmcia é a computação tabular. Uma maneira simples, porém ingênua, de encontrar o valor de $D(i,j)$ seria implementar diretamente a recorrência em alguma linguagem de programação que suporte chamadas recursivas de um método. Entretanto, essa abordagem seria extremamente ineficiente, uma vez que uma mesma instância de um subproblema $D(i,j)$ seria chamada diversas vezes para calcular problemas maiores distintos.

Uma outra abordagem é mais eficiente nesse caso: primeiramente $D(i,j)$ é calculado para os menores valores de i,j possíveis. A partir de então determinam-se $D(i,j)$ para i,j maiores

a partir destes valores menores, evitando que um mesmo $D(i,j)$ seja calculado mais de uma vez. Para isto, utiliza-se a computação tabular. Um matriz de $m+1$ linhas por $n+1$ colunas representa todos os valores de $D(i,j)$ possíveis. Cada linha tem um índice i variando de 0 a m (tamanho da string X) e cada coluna tem um índice j variando de 0 a n (tamanho da string Y). Assim, cada posição (i,j) da matriz corresponde ao valor de $D(i,j)$.

O primeiro passo da computação tabular é preencher a matriz para o caso base $D(i,0) = i$ e $D(0,j) = j$. Isto é feito preenchendo linha $i = 0$ e a coluna $j = 0$ conforme a figura 1.

$D(i,j)$			y_1	y_2	y_3	...	y_n
		0	1	2	3	...	n
	0	0	1	2	3		n
x_1	1	1					
x_2	2	2					
x_3	3	3					
$\vdots$	$\vdots$						
x_m	m	m					

Tabela 1: Matriz de computação tabular preenchida com o caso base $D(i,0) = i$ e $D(0,j) = j$

A partir deste momento a matriz é preenchida por linhas, começando pelo primeiro elemento não preenchido em cada linha e prosseguindo em ordem crescente do valor de j. Em cada momento, quando for preciso calcular um valor de $D(i,j)$, já estarão disponíveis os valores de $D(i-1,j)$, $D(i,j-1)$ e $D(i-1,j-1)$ necessários, e assim a matriz pode ser preenchida com facilidade. A tabela 2 ilustra isso:

$D(i,j)$			y_1	y_2	y_3	...	y_n
		0	1	2	3	...	n
	0	0	1	2	3		n
x_1	1	1	D(1,1)	D(1,2)	D(1,3)		D(1,n)
x_2	2	2	D(2,1)	D(2,2)	D(2,3)		D(2,n)
x_3	3	3	D(3,1)	D(3,2)	D(3,3)		D(3,n)
$\vdots$	$\vdots$					D(i,j)	
x_m	m	m					

Tabela 2: Cálculo de um $D(i,j)$ qualquer utilizando a matriz de computação tabular

Uma vez a matriz completamente preenchida, o valor do elmeneto (n,m) corresponde à distância de edição $D(m,n)$ procurada.

1.1.3 O traceback

Uma vez calculada a distância de edição, é preciso determinar a seqüência de operações (transcrição) que permite transormar X em Y com esse número mínimo de operações. Uma maneira simples de fazer isso é atribuir ponteiros a cada elemento (i,j) da matriz da seguinte forma:

- Atribuir um ponteiro da célula (i,j) para a célula $(i,j-1)$ se $D(i,j) = D(i,j-1) + 1$.

- Atribuir um ponteiro da célula (i, j) para a célula $(i-1, j)$ se $D(i, j) = D(i-1, j) + 1$.
- Atribuir um ponteiro da célula (i, j) para a célula $(i-1, j-1)$ se $D(i, j) = D(i-1, j-1) + t(i, j)$.

Essa regra também se aplica à linha $i = 0$ (em que todos os ponteiros apontam para a esquerda) e à coluna $j = 0$ (em que todos os ponteiros apontam para cima). É possível que alguns dos demais elementos da matriz possam ter ponteiros para mais de outro elemento. É possível determinar a seqüência de operações que transforam X em Y seguindo através dos ponteiros um caminho de (m, n) até $(0, 0)$. A interpretação do significado dos ponteiros é a seguinte: um ponteiro horizontal de (i, j) para a célula $(i, j-1)$ indica uma inserção I do caractere $Y(j)$ em X; um ponteiro vertical de (i, j) para a célula $(i-1, j)$ indica uma deleção D do caractere $X(i)$ de X; um ponteiro diagonal de (i, j) para a célula $(i-1, j-1)$ indica um casamento M se $X(i) = Y(j)$ ou uma substituição R se $X(i) \neq Y(j)$.

Qualquer caminho de (m, n) até $(0, 0)$ representa uma seqüência de operações ótima (não necessariamente única) para a transformação de X em Y. Pela maneira como a matriz foi construída, existe sempre pelo menos um caminho desta forma, e seu custo é igual a $D(i, j)$.

1.1.4 Pseudo-código para o algoritmo

O algoritmo 1 apresenta em alto nível o método de programação dinâmica implementado para o problema da distância de edição.

1.2 Análise de complexidade

Nesta seção são apresentadas as análises de complexidade de tempo e de espaço do método proposto.

1.2.1 Complexidade de tempo

O algoritmo precisa calcular cada elemento da matriz individualmente, e este cálculo é feito com um número constante de operações. Como o número de elementos a ser calculado é $(m+1) \cdot (n+1)$, a complexidade de tempo da computação tabular é da ordem de $O(mn)$. O passo de *traceback*, subseqüente à computação tabular, gasta no máximo $O(m+n)$ operações para encontrar uma transcrição que transforme X em Y (pois o maior caminho que pode ser formado pelos ponteiros da tabela é um que tenha exatamente n setas para a esquerda e m setas para cima). Logo, a complexidade que domina o algoritmo como um todo é a da computação tabular, e o algoritmo tem complexidade de tempo de $O(mn)$.

1.2.2 Complexidade de espaço

A maior estrutura de dados armazenada em memória para a execução do algoritmo é a matriz utilizada para a computação tabular. Essa matriz possui $n+1$ linhas por $m+1$ colunas, sendo que cada elemento da matriz possui tamanho constante. Portanto, a complexidade de espaço desta matriz, e do algoritmo como um todo, é da ordem de $O(mn)$.

1.3 Implementação

O algoritmo foi implementado em linguagem C e os fontes encontram-se em anexo.

Algoritmo 1 Método de programação dinâmica para a distância de edição

Require: $X[1..m], Y[1..n]$
  {A string X deve ser convertida na string Y.}
Ensure: $ed(X, Y), T$
  {A distância de edição e uma transcrição T de X para Y}
  {Passo base}
  for $i = 0$ to m **do**
    $D[i][0] \leftarrow i$ {Passo base $D(i, 0) = i$}
  end for
  for $j = 0$ to n **do**
    $D[0][j] \leftarrow i$ {Passo base $D(0, j) = j$}
  end for
  {Computação Tabular}
  for $i = 0$ to m **do**
    for $j = 0$ to n **do**
      {Calcula $t(i, j)$}
      if $X(i) = Y(j)$ **then**
        $t(i, j) \leftarrow 0$
      else
        $t(i, j) \leftarrow 1$
      end if
      {Recursão sobre $D(i, j)$}
      $D(i, j) \leftarrow min[D(i - 1, j) + 1, D(i, j - 1) + 1, D(i - 1, j - 1) + t(i, j)]$
      {Atribui apontadores para esta célula}
      if $D(i, j) = D(i - 1, j) + 1$ **then**
        $D(i, j).antecessor \leftarrow ESQUERDA$ {Houve inserção}
      end if
      if $D(i, j) = D(i, j - 1) + 1$ **then**
        $D(i, j).antecessor \leftarrow CIMA$ {Houve deleção}
      end if
      if $D(i, j) = D(i - 1, j - 1) + t(i, j)$ **then**
        $D(i, j).antecessor \leftarrow DIAGONAL$ {Houve casamento ou troca}
      end if
    end for
  end for
  {Traceback}
  $celula \leftarrow D(m, n)$
  {Encontra uma transcrição possível.}
  $T \leftarrow \emptyset$
  while $celula \neq D(0, 0)$ **do**
    $T \leftarrow T+$ operacao realizada $(I, D, MouR)$ de acordo com o antecessor
    $celula \leftarrow celula.antecessor$
  end while
  $T \leftarrow inversa(T)$ {Coloca a seqüência de operações na ordem correta.}

1.4 Exemplo de funcionamento

Nesta seção será mostrado o funcionamento do algoritmo par ao exemplo `matranda` e `saturadas`.

Conforme o algoritmo 1, o primeiro passo do programa é montar a matriz de computação tabular para as strings $X =$ `matranda` e $Y =$ `saturadas`. Em seguida já é possível preencher a primeira linha da matriz com os valores do passo base $D(0, j) = j$ e a primeira coluna com os valores $D(i, 0) = i$. Além disso os ponteiros também já são colocados, conforme mostra a tabela 3.

$D(i,j)$			s	a	t	u	r	a	d	a	s
		0	1	2	3	4	5	6	7	8	9
	0	0	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$	$\leftarrow 5$	$\leftarrow 6$	$\leftarrow 7$	$\leftarrow 8$	$\leftarrow 9$
m	1	$\uparrow 1$									
a	2	$\uparrow 2$									
t	3	$\uparrow 3$									
r	4	$\uparrow 4$									
a	5	$\uparrow 5$									
n	6	$\uparrow 6$									
d	7	$\uparrow 7$									
a	8	$\uparrow 8$									

Tabela 3: Passo base para o exemplo `matranda` e `saturadas`

Após o passo base começa a computação tabular, calculando para cada linha o valor de $D(i, j)$ em ordem crescente de j. O primeiro elemento a ser computado é o $D(1, 1) = min[D(0, 1) + 1, D(1, 0) + 1, D(0, 0) + 1] = 1$. Além disso, como $D(1, 1) = D(0, 0) + 1$, a célula $D(1, 1)$ recebe uma seta para a diagonal. O próximo passo é calcular $D(1, 2) = min[D(0, 2) + 1, D(1, 1) + 1, D(0, 1) + 0] = 2$. Além disso, como $D(1, 1) = D(1, 1) + 1 = D(0, 1) + 1$, a célula $D(1, 2)$ recebe uma seta para a esquerda e outra para a diagonal, conforme a tabela 4.

$D(i,j)$			s	a	t	u	r	a	d	a	s
		0	1	2	3	4	5	6	7	8	9
	0	0	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$	$\leftarrow 5$	$\leftarrow 6$	$\leftarrow 7$	$\leftarrow 8$	$\leftarrow 9$
m	1	$\uparrow 1$	$\nwarrow 1$	$\leftarrow\nwarrow 2$	$\leftarrow\nwarrow 3$	$\leftarrow\nwarrow 4$	$\leftarrow\nwarrow 5$	$\leftarrow\nwarrow 6$	$\leftarrow\nwarrow 7$	$\leftarrow\nwarrow 8$	$\leftarrow\nwarrow 9$
a	2	$\uparrow 2$	$\nwarrow\uparrow 2$	$\nwarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$	$\leftarrow\nwarrow 5$	$\leftarrow 6$	$\leftarrow\nwarrow 7$	$\leftarrow 8$
t	3	$\uparrow 3$	$\nwarrow\uparrow 3$	$\uparrow 2$	$\nwarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$	$\leftarrow 5$	$\leftarrow 6$	$\leftarrow 7$
r	4	$\uparrow 4$	$\nwarrow\uparrow 4$	$\uparrow 3$	$\uparrow 2$	$\nwarrow 2$	$\nwarrow 2$	$\leftarrow 3$	$\leftarrow 4$	$\leftarrow 5$	$\leftarrow 6$
a	5	$\uparrow 5$	$\nwarrow\uparrow 5$	$\nwarrow\uparrow 4$	$\uparrow 3$	$\nwarrow\uparrow 3$	$\nwarrow\uparrow 3$	$\nwarrow 2$	$\leftarrow 3$	$\leftarrow\nwarrow 4$	$\leftarrow 5$
n	6	$\uparrow 6$	$\nwarrow\uparrow 6$	$\uparrow 5$	$\uparrow 4$	$\nwarrow\uparrow 4$	$\nwarrow\uparrow 4$	$\uparrow 3$	$\nwarrow 3$	$\leftarrow\nwarrow 4$	$\leftarrow\nwarrow 5$
d	7	$\uparrow 7$	$\nwarrow\uparrow 7$	$\uparrow 6$	$\uparrow 5$	$\nwarrow\uparrow 5$	$\nwarrow\uparrow 5$	$\uparrow 4$	$\nwarrow 3$	$\leftarrow\nwarrow 4$	$\leftarrow\nwarrow 5$
a	8	$\uparrow 8$	$\nwarrow\uparrow 8$	$\nwarrow\uparrow 7$	$\uparrow 6$	$\nwarrow\uparrow 6$	$\nwarrow\uparrow 6$	$\nwarrow\uparrow 5$	$\uparrow 4$	$\nwarrow 3$	$\leftarrow 4$

Tabela 4: Matriz final para o exemplo `matranda` e `saturadas`

Após o preenchimento da tabela, o algoritmo já sabe que a distância de edição entre $X =$ `matranda` e $Y =$ `saturadas` é o valor da célula $D(m, n) = D(8, 9) = 4$.

Neste ponto pode-se iniciar o *traceback* para recuperar as operações realizadas. Começando

de $D(8,9)$, o algoritmo vê que essa célula aponta para a esquerda, o que equivale a uma inserção I do caractere $Y(9) =$ `s` na primeira string. O programa passa então para a célula apontada, ou seja, $D(8,8)$, e descobre que ela aponta para a diagonal. Como $X(8) = Y(8) =$ `a` o algoritmo sabe que houve um casamento M entre essas letras e então se move para a célula apontada $D(7,7)$. O processo prossegue até ser atingida a célula $D(0,0)$, quando o *traceback* obtém a transcrição completa (depois de já colocada na ordem correta, de frente para trás): $RMMIMMDMMI$. Essa transcrição implica que para transformar a string $X =$ `matranda` em $Y =$ `saturadas`, deve-se:

- começar com a string original: `matranda`
- $R \rightarrow$ substituir `m` por `s` $\rightarrow$ `Satranda`
- $M \rightarrow$ casar `a` com `a`: $\rightarrow$ `SAtranda`
- $M \rightarrow$ casar `t` com `t`: $\rightarrow$ `SATranda`
- $I \rightarrow$ inserir `u`: $\rightarrow$ `SATUranda`
- $M \rightarrow$ casar `r` com `r`: $\rightarrow$ `SATURanda`
- $M \rightarrow$ casar `a` com `a`: $\rightarrow$ `SATURAnda`
- $D \rightarrow$ deletar `n`: : $\rightarrow$ `SATURAda`
- $M \rightarrow$ casar `d` com `d`: $\rightarrow$ `SATURADa`
- $M \rightarrow$ casar `a` com `a`: $\rightarrow$ `SATURADA`
- $I \rightarrow$ inserir `s`: $\rightarrow$ `SATURADAS`

E assim termina a execução do algoritmo, retornando a distância de edição 4 e a transrição $RMMIMMDMMI$.

2 Conjunto Independente Máximo

Dado um grafo $G = (V, A)$, um *conjunto independente de vértices* é um subconjunto $V' \subseteq V$ tal que todo par de vértices de V' não é adjacente (isté, se $x, y \in V'$, então a aresta $x, y \notin A$).

Um *conjunto independente maximal* [2] é máximo se todos os outros cojuntos independentes têm cardinalidade menor ou igual. O cojunto $\{2,3\}$ na Figura não é um conjunto independente maximal enquanto os cojuntos $0, 1$ e $\{2,3,4\}$ são conjuntos independentes maximais, sendo que o cojunto $\{2,3,4\}$ é um cojunto independente máximo.

Esse problema tem várias aplicações práticas. Exemplos:

1. 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 cojunto independente máximo representa o maior cojunto de pessoas que não se conhecem.

[2] Um cojnunto independente é *maximal* quando não existe nenhum outro conjunto idependente que o contenha, isté, um cojunto que não pode ser completado.

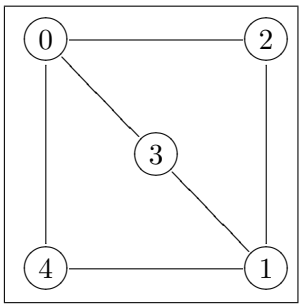

Figura 1: Exemplo de conjunto independente maximal.

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

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

2.1 Prova de que o conjunto independente é $\mathcal{NP}$-Completo

Teorema 2. *O problema de se determinar se um grafo $G = (V, A)$ possui um conjunto independente de vértices I tal que $|I| > k$ para uma constante k é $\mathcal{NP}$-Completo.*

Prova: Em primeiro lugar, o problema do conjunto independente de vértices está em $\mathcal{NP}$ (o algoritmo 2 verifica se um conjunto S de vértices é independente e maior que k em tempo polinomial).

Em segundo lugar mostramos que o *CNF-SAT* (sabidamente $\mathcal{NP}$-Completo) se reduz ao problema do conjunto independente de vértices. Seja uma instância do *SAT* na forma normal conjuntiva[3]. Vamos criar um grafo G a partir desta instância. Primeiramente, criamos um vértice para cada literal na fórmula, incluindo vértices duplicados para aqueles literais que ocorrem mais de uma vez. Colocamos uma aresta entre dois literais quaisquer que sejam a negação um do outro ou dois literais quaisquer que estejam na mesma cláusula. Agora vamos mostrar que G tem um conjunto independente de tamanho pelo menos k, onde k é o número de cláusulas se, e somente se, a fórmula original é satisfatível.

Suponha que temos uma atribuição de valores aos literais que satisfaça a fórmula original. Então podemos escolher um literal de cada cláusula ao qual foi atribuído o valor verdadeiro.

[3]Na forma normal conjuntiva uma expressão lógica é dada pela conjunção de cláusulas, sendo cada cláusula uma série de zero ou mais disjunções.

Algoritmo 2 Verificação de solução do problema

```
Require: G = (V, A), I
  for i ← 0 até |I| do
    for j ← i até |I| do
      if existe aresta (i, j) then
        retorne FALHA
      end if
    end for
  end for
  if |I| > k then
    retorne SUCESSO
  end if
```

Este conjunto é independente: ele contém apenas um literal de cada cláusula e nenhuma atribuição faz um literal e sua negação verdadeiros ao mesmo tempo.

Agora suponha que temos um conjunto independente de tamanho k ou maior. Este conjunto não pode conter dois literais da mesma cláusula, já que eles seriam adjacentes. Mas então já que há pelo menos k vértices e há k cláusulas, devemos ter pelo menos um vértice em cada cláusula. Não é possível que o conjunto tenha um literal e sua negação, porque há arestas entre eles. Isso significa que é fácil escolher uma atribuição que faça todos esses k literais verdadeiros, e essa atribuição satisfaz a fórmula original. ∎

2.2 Algoritmo para a solução ótima

Neste trabalho foram implementados dois algoritmos distintos para encontrar a solução ótima para o problema do conjunto independente máximo. O primeiro algoritmo, chamado de "*Naive*" [4] utiliza simplesmente força bruta e, apesar de apresentar algum nível de refinamento, ainda é extremamente ineficiente. O segundo algoritmo, chamado de *"Bron-Kerbosch"* [2], se vale de um conhecimento maior sobre a estrutura do problema do conjunto independente máximo para obter um algoritmo significativamente mais eficiente que o primeiro. No entanto, mesmo o segundo algoritmo não apresenta uma performance razoavelmente aceitável à medida em que o tamanho da entrada aumenta (o tempo de execução no pior caso ainda é exponencial).

2.2.1 Algoritmo *Naive*

O primeiro algoritmo implementado utiliza simplesmente força bruta, daí o nome de *Naive* (inocente). Basicamente, todas as combinações possíveis de subconjuntos de vértices são geradas, sendo que cada subconjunto é testado para verificar se é independente. Após todos os subconjuntos terem sido testados, o maior deles é retornado como o conjunto independente máximo (I_{max}). Essa primeira idéia é apresentada no algoritmo 3.

Esta primeira abordagem é extremamente ineficiente mesmo para grafos não muito grandes (como com 20 ou 30 vértices). O problema é que é preciso testar *todos* os subconjuntos $S \in V$ antes de que se possa dizer qual deles é o I_{max}. Uma alteração simples que melhora ligeiramente este algoritmo é testar os conjuntos S em ordem decrescente de tamanho, começando

[4] Inocente

Algoritmo 3 Naive 1

Require: $G = (V, A)$
Ensure: $I_{max}(G)$
  $I_{max} \leftarrow \emptyset$
  for all $S \in V$ **do**
    if *independente*(S) **then**
      if $|S| > |I_{max}|$ **then**
        $I_{max} \leftarrow S$
      end if
    end if
  end for

por $S = V$ (testando se o grafo inteiro não é um conjunto independente). Dessa forma, assim que um conjunto for testado e verificado ser independente, o algoritmo pode parar, pois com certeza não há outro conjunto maximal maior que ele. Esta melhoria é apresentada no algoritmo 4.

Algoritmo 4 Naive 2

Require: $G = (V, A)$
Ensure: $I_{max}(G)$
  for $k = |V|$ até 1 **do**
    for all $S \in V$ tal que $|S| = k$ **do**
      if *independente*(S) **then**
        $I_{max} \leftarrow S$
        Pare.
      end if
    end for
  end for

Ainda é possível fazer outra melhoria no algoritmo 4. Para isto será utilizado o resultado do teorema 3.

Teorema 3. *Seja I_{max} um conjunto independente máximo do grafo simples*[5] $G = (V, A)$. *Seja $n = |I_{max}|$. Então*

$$n \leq \left\lfloor \frac{1 + \sqrt{1 - 4(2|A| - |V|^2 + |V|)}}{2} \right\rfloor$$

Prova: Seja S um conjunto independente de vértices qualquer (não necessariamente maximal) de $G = (V, A)$, tal que $|S| = m$. O máximo de arestas que um grafo simples pode possuir é $|V|(|V| - 1)/2$. Mas como S é independente, então as $m(m - 1)/2$ arestas possíveis entre seus vértices não podem pertencer ao grafo. Ou seja, o número de arestas $|A|$ não pode ser maior que $|V|(|V| - 1)/2 - m(m - 1)/2$. Desenvolvendo a fórmula:

[5]Um grafo simples não possui arestas duplas nem self-loops.

$$|A| \leq |V|(|V|-1)/2 - m(m-1)/2 \implies$$

$$m^2 - m + (2|A| - |V|^2 + |V| \leq 0 \implies \tag{4}$$

$$\frac{1-\sqrt{1-4(2|A|-|V|^2+|V|)}}{2} \leq m \leq \frac{1+\sqrt{1-4(2|A|-|V|^2+|V|)}}{2}$$

Como m é inteiro e não pode ser menor que 0:

$$m \leq \left\lfloor \frac{1+\sqrt{1-4(2|A|-|V|^2+|V|)}}{2} \right\rfloor \tag{5}$$

Como a equação 5 vale para qualquer m, em particular ela vale também para o caso em que $m = n$, ou seja, o caso do conjunto independente máximo I_{max}. Dessa forma $\left\lfloor \frac{1+\sqrt{1-4(2|A|-|V|^2+|V|)}}{2} \right\rfloor$ é um limite superior para o tamanho do conjunto idependente que o grafo pode possuir. ∎

O resultado do teorema 3 pode ser utilizado para melhorar o aloritmo *Naive 2*. Basta não testar subconjuntos de vértices maiores que o limite superior demonstrado. O algoritmo 5 apresenta esta melhoria.

Algoritmo 5 Naive 3

Require: $G = (V, A)$
Ensure: $I_{max}(G)$
  for $k = \left\lfloor \frac{1+\sqrt{1-4(2|A|-|V|^2+|V|)}}{2} \right\rfloor$ até 1 **do**
    for all $S \in V$ tal que $|S| = k$ **do**
      if *independente*(S) **then**
        $I_{max} \leftarrow S$
        Pare.
      end if
    end for
  end for

Neste trabalho foi implementada a terceira versão do algoritmo *Naive*. O código fonte em C encontra-se em anexo.

Foi realizada uma bateria de testes com a implementação, com a seguinte especificação:

- Os tamanhos dos grafos variaram de 5 a 25 vértices, variando de 5 em 5.
- Foram gerados 33 grafos de cada tamanho.
- Cada grafo gerado de tamanho $|V|$ possuía entre $|V|-1 \leq |E| \leq |V|(|V|-1)/2$ arestas, sendo que o número de arestas foi randomicamente gerado. Com isso espera-se obter grafos aleatórios com conectividade[6] variável.
- O algoritmo *Naive* foi aplicado a estes grafos.

[6] A conectividade de um grafo, ou o grau médio de seus vértices, é uma medida de quão denso é o grafo.

Como um primeiro resultado do experimento, o algoritmo se mostrou muito ineficiente quando o tamanho do grafo de entrada aumenta. A figura 2 apresenta um gráfico do desempenho do algoritmo (tempo x tamanho $|V|$ do grafo). O tempo indicado no gráfico é o tempo médio das 33 execuções para cada tamanho do grafo.

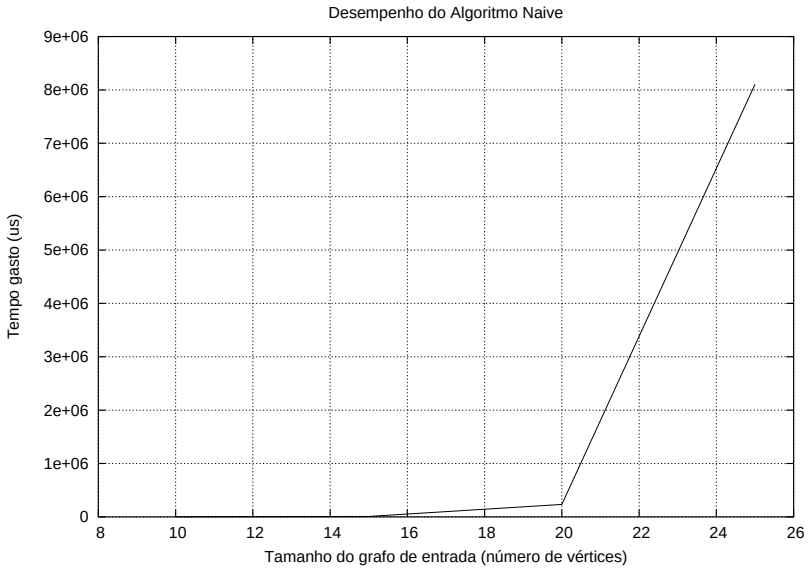

Figura 2: Desempenho (tempo x tamanho da entrada) para algoritmo *Naive*.

A curva da figura demonstra claramente a natureza exponencial do algoritmo. Para grafos de tamanho até $|V| = 25$ o *Naive* obteve uma solução em um tempo médio razoável. Para estes grafos o tempo médio de execução foi de $8.1s$. Como no pior caso o comportamento do algoritmo é de $O(2^{|V|})$ (pois todos os subconjuntos de vértices são testados), se a entrada aumentasse em 10 vezes o tempo de execução seria elevado à décima potência, ou seja, $8.1^{10}s \approx 1215766545s \approx 38anos$ (um tempo *muito maior*).

2.2.2 Algoritmo *Bron-Kerbosch*

Um algoritmo ótimo mais elaborado foi proposto por Coen Bron e Joep Kerbosch em 1971 [2]. A base do algoritmo ainda é a força bruta, mas utiliza-se um conhecimento maior sobre a natureza do problema do conjunto independente máximo.

Uma abordagem de força bruta simples seria gerar todas os subconjuntos de vértices possíveis para o grafo de entrada e então testar se o subconjunto gerado é um conjunto independente de vértices. Durante o processo os maiores conjuntos encontrados são armazenados. O problema deste método é que ele se torna muito ineficiente para grandes grafos de entrada (grafos com 100 ou mais vértices já se tornam praticamente intratáveis). Essa ineficiência se deve ao fato de que um grande número de conjuntos independentes gerados é rejeitado porque mais tarde se descobre que existe outro conjunto independente maior que os contém. A grande vantagem do algoritmo *Bron-Kerbosch* é que ele gera apenas conjuntos independentes maximais, evitando assim que cada conjunto gerado tenha que ser comparado com os previamente testados.

Uma discussão detalhada de como o algoritmo funciona (e porque funciona) pode ser encontrada em [3]. Essencialmente o algoritmo consiste em uma árvore de busca enumerativa

na qual, num dado estágio k, um conjunto independente de vértices S_k é aumentado pela adição de outro vértice disponível para produzir um novo conjunto independente S_{k+1} num estágio $k+1$. O processo continua até que nenhum outro aumento do conjunto seja possivel e ele se torne maximal.

Num estágio k, seja Q_k o maior conjunto de vértices para o qual $S_k \cap Q_k = \emptyset$, ou seja, qualquer vértice de Q_k adicionado a S_k produz um conjunto S_{k+1} que também é um conjunto independente.

Em qualquer ponto do algoritmo, o conjutno Q_k possui dois tipos de vértices: vértices em Q_k^- que já foram usados em uma tentativa prévia de aumentar S_k, e vértices em Q_k^+ ainda não foram usados. Uma ramificação da árvore consiste em escolher um vértice $x_{i.k} \in Q_k^+$ e adicioná-lo a S_k para produzir:

$$S_{k+1} = S_k \cup \{x_{i.k}\} \tag{6}$$

e criando os novos conjuntos:

$$Q_{k+1}^- = Q_k^- - adjacentes(x_{i.k}) \tag{7}$$

e

$$Q_{k+1}^+ = Q_k^+ - adjacentes(x_{i.k}) - \{x_{i.k}\} \tag{8}$$

Um passo de *backtracking* [7] do algoritmo envolve a remoção de $x_{i.k}$ de S_{k+1} para convertê-lo novamente no antigo S_k e então remover $x_{i.k}$ do antigo conjunto Q_k^+ e adicioná-lo ao antigo Q_k^- para formar os novos subconjuntos Q_k^+ e Q_k^-.

O conjunto S_k é maximal se, e somente se, $Q_k^+ = \emptyset$ (não há nenhum vértice novo disponível para aumentar o conjunto) e $Q_k^- = \emptyset$ (não houve em algum passo anterior um conjunto maior que contivesse o atual). Essas condições são expressas por:

$$Q_k^+ = Q_k^- = \emptyset \tag{9}$$

Como Q_k^- só é diminuído quando um vértice $x_{i.k} \in Q_k^+$ que tenha adjacentes em Q_k^- é escolhido para entrar em S_k (pela equação 7), conclui-se que no momento em que um vértice de $x \in Q_k^-$ não tiver mais adjacentes em Q_k^+, ele não poderá mais sair de $x \in Q_k^-$ e este conjunto nunca esvaziará (a condição 9 nunca será satisfeita). Ou seja, pode-se parar de expandir a árvore de busca quando a seguinte condição for satisfeita:

$$\exists x \in Q_k^- \; tal \; que \; adjacentes(x) \cap Q_k^+ = \emptyset \tag{10}$$

Com o que foi dito até agora, já é possível implementar o algoritmo *Bron-Kerbosch*. No entanto, uma outra melhoria ainda pode ser feita, escolhendo-se um em cada iteração k um elemento x para aumentar S_k que force a condição 10 o mais rapidamente possível. Para determinar como será a escolha de tal x, é preciso definir a função $\Delta(x)$:

$$\Delta(x) = |adjacentes(x) \cap Q_k^+| \tag{11}$$

[7] Uma tradução para o termo em inglês *backtracking* seria "retorno". Neste trabalho optou-se por não traduzir o termo.

Seja $x* \in Q_k^+$ o elemento que minimiza a função $\Delta(x)$. Para forçar a condição 10 o mais rapidamente possível, deve-se escolher um elemento x que seja adjacente a $x*$ e que pertença a Q_k^+, ou seja:

$$x = adjacentes(x*) \cap Q_k^+ \tag{12}$$

O algoritmo 6 apresenta o pseudo-código que implementa o que foi discutido até o momento. Uma implementação deste algoritmo em *ALGOL* encontra-se em [2]. Neste trabalho a implementado foi realizada utilizando a linguagem *C*, mas alterando o algoritmo original para armazenar somente o maior conjunto independente maximal encontrado, ou seja, o conjunto independente máximo. O código fonte encontra-se em anexo.

Foi realizada uma bateria de testes em cima desta implementação, com a seguinte especificação:

- Os tamanhos dos grafos variaram de 10 a 100 vértices, variando de 5 em 5.
- Foram gerados 33 grafos de cada tamanho.
- Cada grafo gerado de tamanho $|V|$ possuíaa entre $|V|-1 \leq |E| \leq |V|(|V|-1)/2$ arestas, sendo que o número de arestas foi randomicamente gerado. Com isso espera-se obter grafos aleatórios com conectividade variável.
- O algoritmo *Bron-Kerbosch* foi aplicado a estes grafos.

O algoritmo se mostrou muito mais eficiente que o *Naive*. Comparando o tempo médio para um grafo de 25 vértices, o algoritmo *Naive* gastou $8.1s$, enquanto o *Bron-Kerbosch* gastou apenas $0,075s$. A figura 3 mostra o comportamento do algoritmo quando o tamanho da entrada cresce.

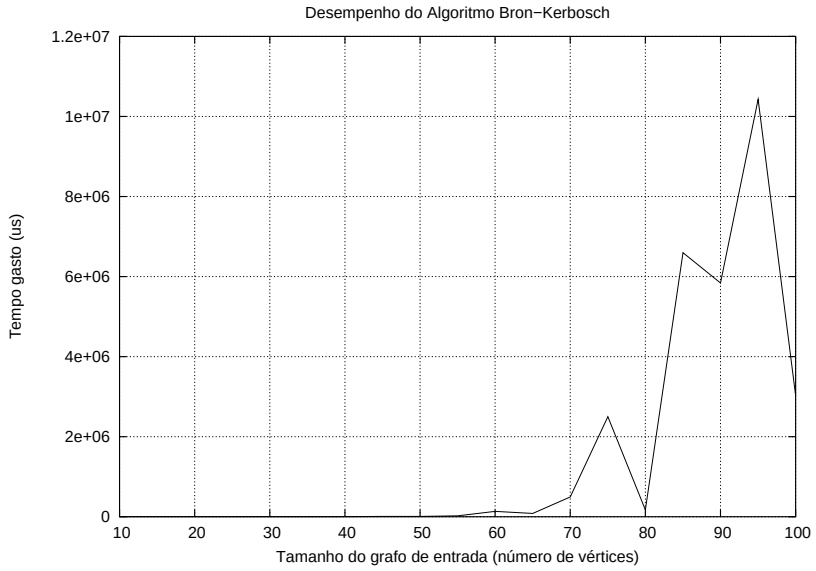

Figura 3: Desempenho (tempo x tamanho da entrada) para algoritmo *Bron-Kerbosch*.

Uma análise do gráfico mostra que o algoritmo foi capaz de encontrar um conjunto independente de vértices máximo para grafos relativamente grandes ($|V| \approx 100$) em um tempo

Algoritmo 6 Bron-Kerbosch

Passo 1: {Inicialização}
$S_0 \leftarrow \emptyset$
$Q_0^- \leftarrow \emptyset$
$Q_0^+ \leftarrow V$
$k \leftarrow 0$
Passo 2: {Ramificação da árvore}
Escolha um vértice $x_{i.k} \in Q_k^+$ que satisfaça a condiçao 12.
$Q_{k+1}^+ \leftarrow Q_k^+ - adjacentes(x_{i.k}) - \{x_{i.k}\}$
$Q_{k+1}^- \leftarrow Q_k^- - adjacentes(x_{i.k})$
Mantenha Q_k^+ e Q_k^- intactos.
$k \leftarrow k + 1$.
Passo 3: {Teste}
if $\exists x \in Q_k^-$ tal que $Q_k^+ \cap adjacentes(x) = \emptyset$ **then**
  Vá para o passo 5.
else
  Vá para o passo 4.
end if
Passo 4:
if $Q_k^- = \emptyset \wedge Q_k^+ = \emptyset$ **then**
  Imprima o conjunto independente maximal S_k.
  Vá para o passo 5.
end if
if $Q_k^+ = \emptyset \wedge Q_k^- \neq \emptyset$ **then**
  Vá para o passo 5.
else
  Vá para o passo 2.
end if
Passo 5: {Backtrack}
$k \leftarrow k - 1$
$S_k \leftarrow S_{k+1} - \{x_{i.k}\}$
Recupere Q_k^+ e Q_k^-.
$Q_k^+ \leftarrow Q_k^+ - \{x_{i.k}\}$
$Q_k^- \leftarrow Q_k^- \cup \{x_{i.k}\}$
if $k = 0 \wedge Q_0^+ = \emptyset$ **then**
  Pare.
else
  Vá para o passo 3.
end if

razoável, cerca de apenas $3s$ (tempo de usuário, não de relógio). Na verdade, para os tamanhos testados, o algoritmo chegou a apresentar comportamento médio menor que o exponencial típico, o que permitiu que fossem encontrados conjuntos independentes para grafos muito grandes. Os testes realizados com os maiores grafos encontram-se na tabela 5.

Em primeiro lugar é importante ressaltar que esses melhores casos não representam necessariamente o comportamento do algoritmo no caso geral, uma vez que não foi possível realizar

Tamanho do grafo ($\|V\|$)	Tamanho do conjunto encontrado	Tempo gasto
550	11	$1min$ $30s$ $037ms$
600	12	$37min$ $04s$ $164ms$
750	16	$5h$ $35min$ $17s$ $044ms$
800	21	$1h$ $21min$ $58s$ $618ms$

Tabela 5: Melhores resultados obtidos com o algoritmo *Bron-Kerbosch*

diversos testes com cada tamanho de entrada (um tempo de cerca de 5 horas gasto com um único experimento, como no caso do grafo com 750 vértices, inviabilizou a realização de mais testes no tempo dado para este trabalho). Um fato interessante é que o tempo gasto para resolver os problemas maiores não cresceu sempre com o tamanho do grafo, uma vez que o grafo de tamanho 800 demorou menos tempo para executar que o grafo de tamanho 750. Isso se deve provavelmente a alguma característica que o grafo maior tenha e que foi bem explorada pelo *Bron-Kerbosh* para diminuir significativamente o espaço de busca e encontrar uma solução num tempo tão "curto". Porém, mesmo que esta eficiência do algoritmo para os casos apresentados se deva a uma ou outra característica especial do grafo de entrada, não deixou de ser surpreendente o fato de ele conseguir resolver problemas tão grandes. Um algoritmo tão simples como o *Naive 1* (algoritmo 3), por exemplo, demora um tempo constante para encontra uma solução ótima para todos os grafos de mesmo tamanho, pois não consegue utilizar nenhuma característica do grafo para reduzir o espaço de busca.

Uma conseqüência desta "eficiência"do *Bron-Kerbosch* (pelo menos para grafos de tamanho até 100) foi a utilização deste algoritmo para gerar uma base de testes de comparação para os algoritmos aproximados implementados neste trabalho. Os conjuntos de grafos que foram utilizados para testar esse outros algoritmos foram processados primeiramente pelo *Bron-Kerbosch* para que, conhecendo-se o tamanho da solução ótima, se pudesse melhor avaliar o desempenho dos demais algoritmos.

Para encerrar a discussão sobre o *Bron-Kerbosch*, é preciso lembrar que no pior caso ele ainda apresenta complexidade exponencial $O(2^{|V|})$ (uma vez que em casos extremos todos os subconjuntos de vértices são testados). Se o tamanho da entrada do caso em que o algoritmo gastou mais tempo (750 vértices) fosse multiplicado por 10, o tempo de execução seria elevado a 10, passando de 5,5 horas para aproximadamente $2 \cdot 10^{34}$ anos.

2.3 Redução do problema da clique ao problema do conjunto independente

O teorema 4 mostra que o problema da clique é redutível ao problema do conjunto independente.

Teorema 4. *Seja um grafo $G = (V, A)$. Um subconjunto de vértices $S \in V$ é uma clique em G se, e somente se, S é um conjunto independente em $\overline{G} = (V, \overline{A})$ (a recíproca também é verdadeira). Como caso particular, a clique máxima de G corresponde ao conjunto independente máximo de $\overline{G}$.*

Prova: O teorema segue direto da definição de grafo complementar, já que uma aresta existe em $\overline{G}$ se, e somente se, não existe em G. Portanto, um subconjunto de vértices completamente conectado (clique) $S \in V$ de G é completamente desconectado em $\overline{G}$ (conjunto independente). A prova da recíproca é inteiramente análoga. Como a afirmação vale para

qualquer clique, ela vale para a clique máxima. ∎

O algoritmo 7 gera o grafo $\overline{G}$ complementar a um grafo G em complexidade de tempo polinomial no número de arestas ($O(|A|)$).

Algoritmo 7 Obter o complementar de um grafo

Require: $G = (V, A)$
Ensure: $\overline{G} = (V, \overline{A})$
  $\overline{A} \leftarrow A$
  for all $a \in A$ do **do**
    $\overline{A} \leftarrow \overline{A} - \{a\}$
  end for

A redução completa do problema do problema da clique ao problema do conjunto independente é apresentada no algoritmo 8.

Algoritmo 8 Redução da clique ao conjunto idependente

Require: $G = (V, A)$
Ensure: $S \in V$ tal que S é a clique máxima de G
  $\overline{G} \leftarrow$ obter o grafo complementar (G)
  $S \leftarrow$ conjunto independente máximo $(\overline{G})$

Para o exemplo da figura 1 o funcionamento do algoritmo é o seguinte:

- É gerado o complemento do grafo, obtendo o novo grafo (figura 4).
- O grafo gerado é então dado como entrada para o algoritmo de determinação do conjunto independente máximo *bron-kerbosch*.
- O resultado gerado pelo algoritmo de *Bron-Kerbosch* é a clique procurada. A clique retornada pelo programa é consiste no conjunto de vértices $\{2, 3, 4\}$.

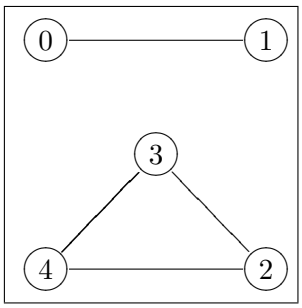

Figura 4: Grafo complementar ao do exemplo.

2.4 Algoritmos aproximados

Nesta seção discute-se a implementação de algoritmos aproximados que resolvem o problema do conjunto independente máximo eficientemente e produzem "boas"soluções sob o ponto de

vista prático. Os algoritmos implementados foram o *Ramsey* [4] e o *Clique-Removal* [4]. No entanto, uma razão de aproximação razoável só pôde ser derivada para o segundo algoritmo.

2.4.1 Algoritmos *Ramsey* e *Clique-Removal*

O algoritmo implementado foi o *clique-removal*, um método de aproximação de conjuntos independentes máximos por exclusão de subgrafos [4]. O algoritmo parte de uma idéia gulosa e a refina até obter uma aproximação razoável.

Definição: Seja $G = (V, A)$ um grafo qualquer. $I(G)$ e $C(G)$ são um conjunto independente e uma clique (não necessariamente maximais) para este grafo.

Definição: Seja I_{max} um conjunto independente máximo de G. Seja $n = |I_{max}|$.

Definição: Seja um vértice $v \in V$. $N(v)$ é o grafo induzido pelos vértices adjacentes a v, enquanto $\overline{N}(v)$ é o grafo induzido pelos vértices não-adjacentes a v.

Uma estratégia gulosa para o problema do conjunto independente consiste em escolher um vértice $v \in V$ qualquer e adicioná-lo a $I(G)$, que começa inicialmente vazio. O próximo passo é eliminar os vértices vizinhos de v e tentar aumentar o conjunto independente com eles. Isso equivale a buscar aumentar $I(G)$ com vértices de $\overline{N}(v)$. O problema de se achar uma clique é dual a este. O algoritmo 9 ilustra essa idéia.

Algoritmo 9 Guloso

Require: $G = (V, E)$
Ensure: $I(G), C(G)$
  Escolha $v \in V$
  $I(G) \leftarrow \{v\} \cup I(\overline{N}(v))$
  $C(G) \leftarrow \{v\} \cup C(N(v))$

O problema desta estratégia gulosa é que, ao procurar aumentar $I(G)$ somente com os vértices não vizinhos a v, um conjunto independente maior pode estar sendo ignorado *entre os vizinhos de* v. Dessa forma, a estratégia pode ser melhorada simplesmente ao se testar também entre os vizinhos de v se há um conjunto independente melhor que entre os não vizinhos. Essa melhoria é ilustrada no algoritmo 10.

Algoritmo 10 Ramsey

Require: $G = (V, E)$
Ensure: $(I(G), C(G))$
  if $G = \emptyset$ **then**
    retorne$(\emptyset, \emptyset)$
  end if
  Escolha $v \in V$
  $(C_1, I_1) \leftarrow$ Ramsey$(N(v))$
  $(C_2, I_2) \leftarrow$ Ramsey$(\overline{N}(v))$
  retorne(maior$(C_1 \cup \{v\}, C_2)$, maior$(I_1, I_2 \cup \{v\})$)

Apesar de melhor que o algoritmo *guloso*, o algoirtmo *Ramsey* ainda não apresenta nenhuma garantia de performance. Usando a teoria de Ramsey, Boppana e Halldórsson [4]

mostram que a clique C e o conjunto independente I retornados por *Ramsey(G)* são tais que $|C| \cdot |I| \geq (\log n)^2/4$. Esse resultado ainda não fornece uma garantia de performance para $|I|$, uma vez que $|C|$ pode ser grande.

A proposta da exclusão de subgrafos é modificar o grafo de forma que, eventualmente, $|C|$ se torne pequeno. Isto é feito chamando-se *Ramsey(G)* e removendo-se (excluindo) a clique C retornada. O algoritmo modificado é o *Clique-Removal* e está apresentado em 11.

Algoritmo 11 Clique-Removal

Require: $G = (V, E)$
Ensure: $I(G)$
  $i \leftarrow 1$
  $(C_i, I_i) \leftarrow Ramsey(G)$
  while $G \neq \emptyset$ **do**
    $G \leftarrow G - C_i$
    $i \leftarrow i + 1$
    $(C_i, I_i) \leftarrow Ramsey(G)$
  end while
  retorne$(max_{j=1}^{i} I_j)$

Neste trabalho foram implementados os algoritmos *Ramsey* e *Clique-Removal*. Os códigos fonte em C encontram-se em anexo.

A seguinte bateria de testes foi realizada com estas implementações:

- Os tamanhos dos grafos variaram de 5 a 100 vértices, variando de 5 em 5.
- Foram gerados 33 grafos de cada tamanho.
- Cada grafo gerado de tamanho $|V|$ possuíaa entre $|V|-1 \leq |E| \leq |V|(|V|-1)/2$ arestas, sendo que o número de arestas foi randomicamente gerado. Com isso espera-se obter grafos aleatórios com conectividade[8] variável.
- Os algoritmos *Ramsey* e *Clique-Removal* foram aplicados a estes grafos.

As figuras 5 e 6 mostram os gráficos de tempo x tamanho da entrada obtidos.

As figuras mostram que ambos os algoritmos são sensíveis ao tamanho da entrada. Uma questão inicial a ser considerada é qual a relação de ganho em tempo que esses algoritmos apresentam em relação ao *Bron-Kerbosch*. A figura 7 apresenta uma comparação entre o desempenho destes algoritmos e o que fornece a solução ótima. O gráfico deixa claro que os algoritmos aproximados são *muito* mais eficientes que o ótimo (quase nem é possível enxergar o desempenho dos aproximados no gráfico, tamanha a diferença apresentada).

Para uma comparação entre o *Ramsey* e o *Clique-Removal*, foram realizados mais testes nos mesmos moldes dos realizados até o momento, mas com grafos de até 1000 vértices de tamanho. Os resultados, apresentados na figura 8 mostra que o segundo algoritmo é muito menos eficiente em termos de tempo que o primeiro. Porém, como será visto mais adiante neste documento, ele apresenta uma razão de aproximação (garantia de eficiência) não-trivial para o problema do conjunto independente máximo.

[8] A conectividade de um grafo, ou o grau médio de seus vértices, é uma medida de quão denso é o grafo.

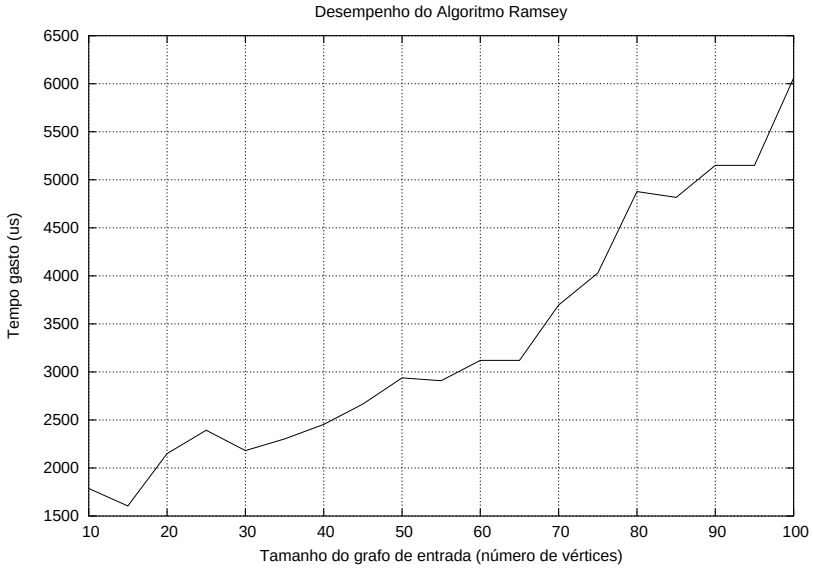

Figura 5: Desempenho (tempo x tamanho da entrada) para algoritmo *Ramsey*.

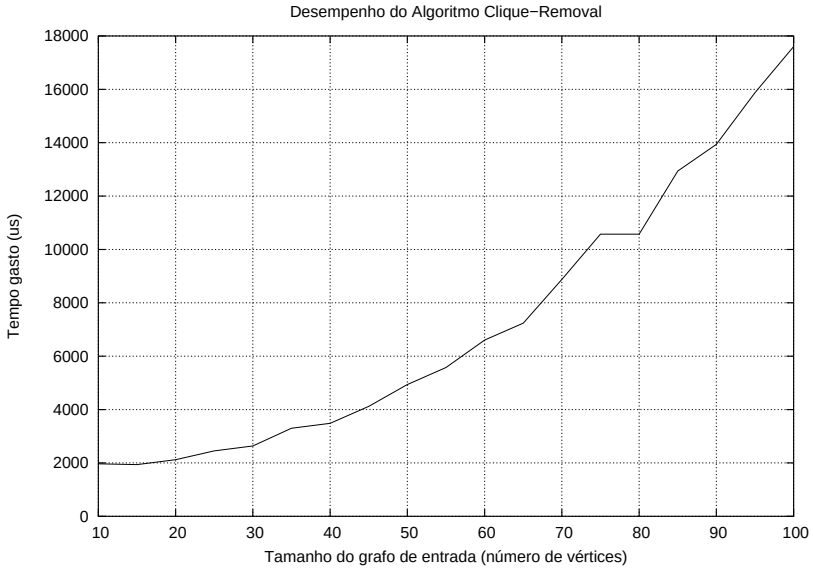

Figura 6: Desempenho (tempo x tamanho da entrada) para algoritmo *Clique-Removal*.

2.4.2 Análise da complexidade de tempo

Em [6] mostra-se que a complexidade de tempo do *Ramsey* é $O(|A|)$. É possível entender (informalmente) essa complexidade pensando que em cada chamada recursiva do algoritmo *Ramsey*, o problema é dividido em dois subproblemas cuja soma dos tamanhos é o tamanho do problema original mais um número de cálculos proporcional ao número de vértices $|V|$ do grafo (o custo de montar a soluçã a partir das sub-soluções calculadas). Dessa forma, a complexidade de tempo do algoritmo é $O(|V|^2)$.

Também em[6] se apresenta que para o algoritmo *Clique-Removal* a complexidade de

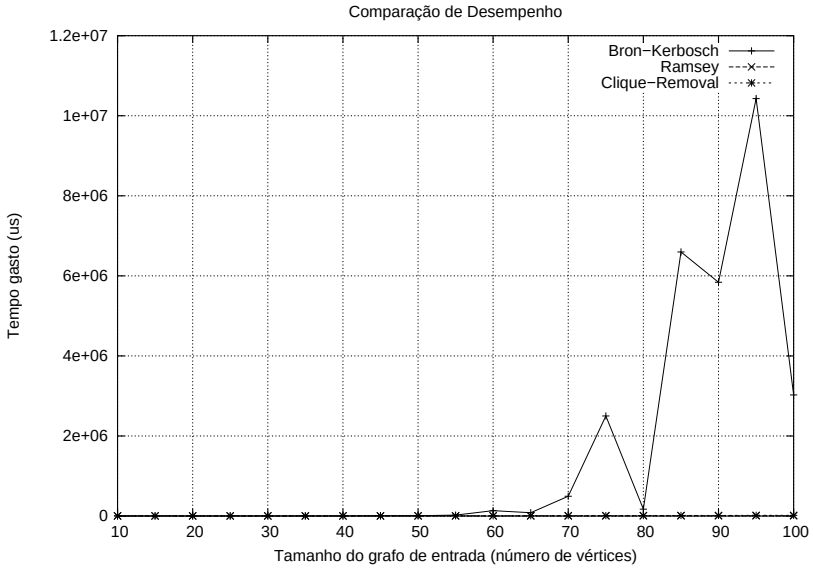

Figura 7: Comparação entre *Bron-Kerbosh*, *Ramsey* e *Clique-Removal*.

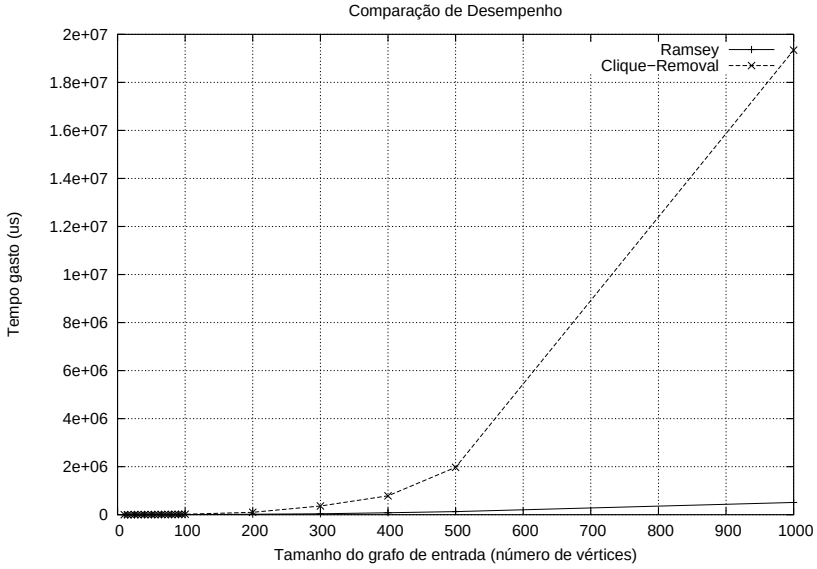

Figura 8: Comparação entre *Ramsey* e *Clique-Removal*.

tempo é de $O(|E||V|\log|V|)$.

2.4.3 Análise da qualidade da aproximação

Ao contrário de muitos outros problemas $\mathcal{NP}$-Difíceis, o problema do conjunto independente ficou por muito tempo sem que os pesquisadores conseguissem construir algoritmos polinomiais com uma garantia de performance não-trivial [5]. O primeiro algoritmo a superar esta dificuldade foi o *Clique-Removal*.

Como um conjunto independente e uma clique em G podem compartilhar no máximo um

vértice em comum, o conjunto independente pode perder no máximo um vértice por iteração do *Clique-Removal*. Se o grafo tiver uma conjunto independente grande o suficiente, uma fração constante do grafo restará mesmo depois que as cliques de um certo tamanho mínimo k forem excluídas. Se *Ramsey* for chamado para o grafo resultante, o tamanho de C pode ser no máximo k. Como discutido em [5], isso implica um limite superior de $|I| \leq 4k/(\log k)^2$. Se o maior conjunto independente for pequeno, a performance do algoritmo é trivialmente razoável. O resultado da análise é uma garantia de performance de $O(n/(\log n)^2)$, onde n é o tamanho do conjunto independente máximo do grafo (mais detalhes podem ser encontrados em [4]).

2.4.4 Testes e resultados empíricos

Para comparar resultados práticos com os teóricos apresentados, foi realizada uma bateria de testes com o algoritmo *Clique-Removal* com as seguintes especificações:

- Os tamanhos dos grafos variaram de 5 a 100 vértices, variando de 5 em 5.
- Foram gerados 33 grafos de cada tamanho.
- Cada grafo gerado de tamanho $|V|$ possuíaa entre $|V|-1 \leq |E| \leq |V|(|V|-1)/2$ arestas, sendo que o número de arestas foi randomicamente gerado. Com isso espera-se obter grafos aleatórios com conectividade variável.
- O algoritmo *Bron-Kerbosch* foi aplicado aos grafos de entrada para se obter o tamanho da solução ótima de cada um.
- O algoritmo *Clique-Removal* também foi aplicado aos grafos de entrada. O tamanho da solução aproximada encontrada em cada caso foi armazenado.

A definição de razão de aproximação diz que, se n é o tamanho da solução ótima e $N*$ é o tamanho de uma solução encontrada pelo algoritmo aproximado, então a razão de aproximação do algoritmo é:

$$max(n/N*) \tag{13}$$

para todo $N*$. Em outras palavra, a razão 13 é uma garantia superior para o tanto que uma solução apresentada pelo algoritmo está afastada da solução ótima. Conforme já discutido neste trabalho, sabe-se que a razão de aproximação para o algoritmo de *Ramsey* é da ordem de $O(n/(\log n)^2)$. Pela definição da notação O, existe uma contante c tal que:

$$c \cdot n/(\log n)^2 \geq n/N* \tag{14}$$

para todo valor de $n/N*$. Como tentativa de determinar empiricamente qual a constante c, para cada solução de tamanho N dada pelo algoritmo *Clique-Removal* para uma instância do problema, a razão $n/N*$ foi calculada. Um gráfico dessa razão em função do tamanho n da solução ótima pode se visto na figura 9, onde cada ponto representa um experimento realizado.

A figura apresenta o ajuste de uma curva da forma $n/(\log n)^2$ sobre os pontos que representam os experimentos de forma a satisfazer a condição 14. A aproximação encontrada foi $c \geq 1.80$. Com esse valor podemos concluir que a constante não pode ser menor que

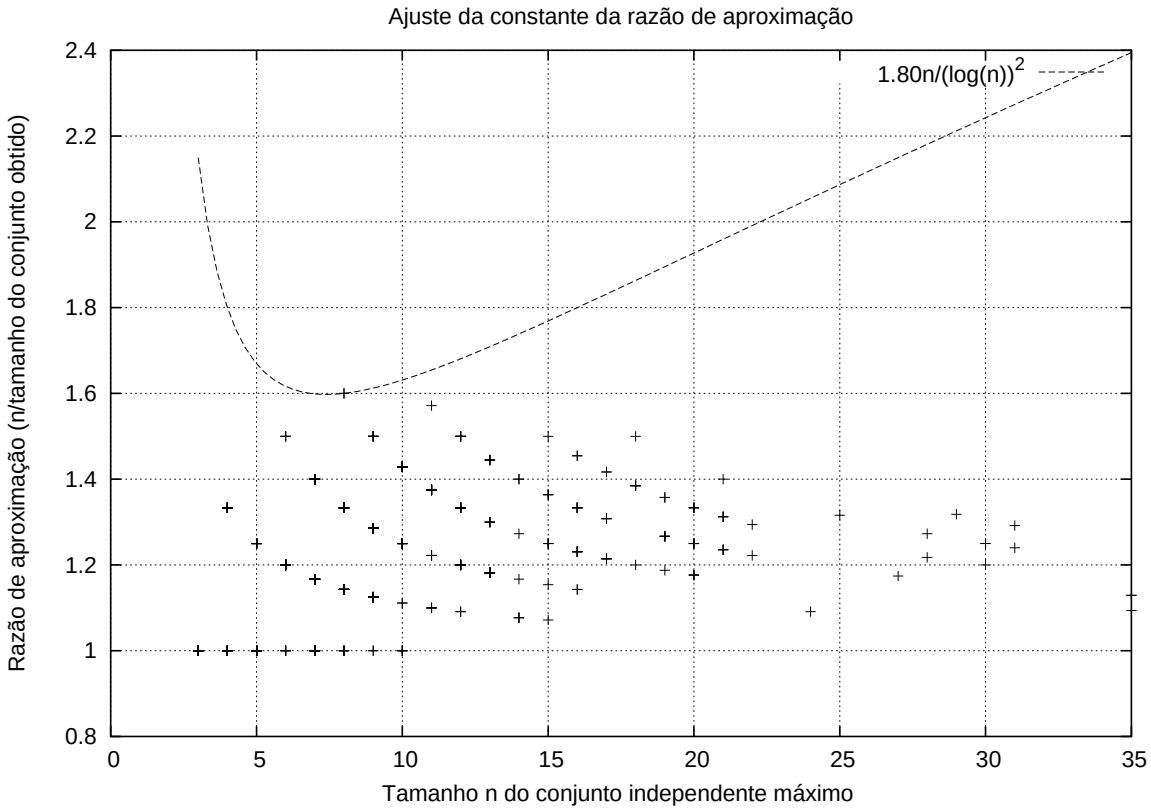

Figura 9: Ajuste empírico da razão de aproximação para o algoritmo *Clique-Removal*

1.80 porque de outra forma o gráfico teria pontos acima desta curva, o que contradiria a equação 14. No entanto, não se pode determinar um limite superior para c, uma vez que a realização de outros experimentos poderia gerar pontos que "forçassem"a curva ainda mais para cima.

A Códigos fonte

A.1 Distância de edição

A.1.1 maindisted.h

```
#ifndef MAINDISTED_H
#define MAINDISTED_H

#define MAX_STRING_LENGTH 1024
#define FALSE 0
#define TRUE 1
#define INSERT 'I'
#define DELETE 'D'
#define REPLACE 'R'
#define MATCH 'M'
#define UNDEFINED 'X'

typedef struct {

int value;
int up;
int left;
int diagonal;

} TCell;

void printCell(TCell c);
void printEdtMatrix(TCell **edtMatrix, int m, int n, char *str1, char *str2);
void readStrings(char *fileName, char str1[], char str2[]);
void minCost(TCell **edtMatrix, int i, int j, char *str1, char *str2);
int *findEdtTranscript(TCell **edtMatrix, int m, int n, char *str1, char *str2);
void printEdtTranscript(int *edtTranscript, int m, int n);
int main(int argc, char *argv[]);

#endif
```

A.1.2 maindisted.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "maindisted.h"

void printCell(TCell c){

printf("%3d[%d%d%d]", c.value, c.left, c.diagonal, c.up);

}

void printEdtMatrix(TCell **edtMatrix, int m, int n, char *str1, char *str2){

int i, j;
```

```
printf("   ");
printf("        ");
for (j=0; j<n; j++) {
printf("    %c   ", str2[j]);
printf(" ");
}
printf("\n");

printf("---");
printf("---------");
for (j=0; j<n; j++) {
printf("---------", str2[j]);
}
printf("\n");

printf(" |");
printf(" ");
for (j=0; j<n+1; j++) {
printCell(edtMatrix[0][j]);
printf(" ");
}
printf("\n");

for (i=1; i<m+1; i++) {

printf("%c|", str1[i-1]);
printf(" ");

for (j=0; j<n+1; j++) {
printCell(edtMatrix[i][j]);
printf(" ");
}
printf("\n");
}

}

void readStrings(char *fileName, char str1[], char str2[]){

FILE *fp;

fp = fopen(fileName, "r");

if (!fp) {
printf("Error: Input file %s not found.\n", fileName);
exit(-1);
}

fscanf(fp, "%s %s", str1, str2);

fclose(fp);
```

```
}


void minCost(TCell **edtMatrix, int i, int j, char *str1, char *str2){

int t;
int leftCost;
int diagonalCost;
int upCost;

if (str1[i-1] == str2[j-1]) {
t = 0;
} else {
t = 1;
}

leftCost = edtMatrix[i][j-1].value + 1;
diagonalCost = edtMatrix[i-1][j-1].value + t;
upCost = edtMatrix[i-1][j].value + 1;

if ( (leftCost <= diagonalCost) && (leftCost <= upCost) ) {
edtMatrix[i][j].value = leftCost;
edtMatrix[i][j].left = TRUE;
}

if ( (diagonalCost <= leftCost) && (diagonalCost <= upCost) ) {
edtMatrix[i][j].value = diagonalCost;
edtMatrix[i][j].diagonal = TRUE;
}

if ( (upCost <= leftCost) && (upCost <= diagonalCost) ) {
edtMatrix[i][j].value = upCost;
edtMatrix[i][j].up = TRUE;
}

}

int *findEdtTranscript(TCell **edtMatrix, int m, int n, char *str1, char *str2){

int i, j;
int k;
int aux;

int *edtTranscript;

edtTranscript = (int*)malloc((m+n)*sizeof(int));
if (!edtTranscript) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}

for(k=0; k<m+n; k++) {
edtTranscript[k] = UNDEFINED;
```

```
}

i = m;
j = n;
k = 0;

while ( (i>0) || (j>0) ) {

if (edtMatrix[i][j].left == TRUE) {
edtTranscript[k] = INSERT;
j--;
} else if (edtMatrix[i][j].diagonal == TRUE) {

if (str1[i-1] == str2[j-1]) {
edtTranscript[k] = MATCH;
} else {
edtTranscript[k] = REPLACE;
}
i--;
j--;
} else if (edtMatrix[i][j].up == TRUE) {
edtTranscript[k] = DELETE;
i--;
}

k++;

}

return(edtTranscript);

}

void printEdtTranscript(int *edtTranscript, int m, int n) {

int k;

for(k=m+n-1; k>=0; k--){

if (edtTranscript[k] != UNDEFINED) {
printf("%c.", edtTranscript[k]);
}
}
printf("\n");

}

void printTPOutput(int *edtTranscript, int m, int n) {

int k;

for(k=0; k<m+n-1; k++){
```

```
switch(edtTranscript[k]) {

case INSERT:
printf("%c ",'1');
break;
case DELETE:
printf("%c ",'2');
break;
case REPLACE:
printf("%c ",'3');
break;
case MATCH:
break;
case UNDEFINED:
break;
break;

}

}

printf("\n");

}


int main(int argc, char *argv[]){

char str1[MAX_STRING_LENGTH];
char str2[MAX_STRING_LENGTH];
char *fileName;
TCell **edtMatrix;
int m, n;
int i, j;
int *edtTranscript;

if (argc < 2) {
printf("Error: the comand line must contain the input file name.\n");
exit(-1);
}

fileName = argv[1];

readStrings(fileName, str1, str2);

m = strlen(str1);
n = strlen(str2);


// Aloca a matriz de edição.
edtMatrix = (TCell**)malloc((m+1) * sizeof(TCell*));

if (!edtMatrix) {
```

```
printf("Error: insuficient memory for this operation!");
exit(-1);
}

for(i=0; i<m+1; i++){

edtMatrix[i] = (TCell*)malloc((n+1) * sizeof(TCell));

if (!edtMatrix[i]) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}

}

// Inicializa a matriz de edição.

for(i=0; i<m+1; i++) {
for(j=0; j<n+1; j++) {
edtMatrix[i][j].value = 0;
edtMatrix[i][j].left = FALSE;
edtMatrix[i][j].diagonal = FALSE;
edtMatrix[i][j].up = FALSE;
}
}

for(j=1; j<n+1; j++) {
edtMatrix[0][j].value = j;
edtMatrix[0][j].left = TRUE;
edtMatrix[0][j].diagonal = FALSE;
edtMatrix[0][j].up = FALSE;
}

for(i=1; i<m+1; i++) {
edtMatrix[i][0].value = i;
edtMatrix[i][0].left = FALSE;
edtMatrix[i][0].diagonal = FALSE;
edtMatrix[i][0].up = TRUE;
}


for(i=1; i<m+1; i++) {
for(j=1; j<n+1; j++) {
minCost(edtMatrix, i, j, str1, str2);
}
}


//printf("str1 = %s \n", str1);
//printf("str2 = %s \n", str2);
//printf("\n");
printEdtMatrix(edtMatrix, m, n, str1, str2);
//printf("\n");
```

```
printf("%d", edtMatrix[m][n].value);
printf("\n");
edtTranscript = findEdtTranscript(edtMatrix, m, n, str1, str2);
//printf("\n");
printEdtTranscript(edtTranscript, m, n);
printTPOutput(edtTranscript, m, n);

return(0);
}
```

A.2 Conjunto independente máximo

A.2.1 mainindep.h

```
#ifndef MAININDEP_H
#define MAININDEP_H

#define ALL_ALGORITHMS "ALL"
#define NAIVE_ALGORITHM "NV"
#define BRON_KERBOSCH_ALGORITHM "BK"
#define RAMSEY_ALGORITHM "RS"
#define CLIQUE_REMOVAL_ALGORITHM "CR"

#define COMPACT_PRINT_MODE "C"
#define DETAILED_PRINT_MODE "D"
#define TP_OUTPUT_PRINT_MODE "TP"

#endif
```

A.2.2 mainindep.c

```
#include "module.h"
#include "graph.h"
#include "naive.h"
#include "bron_kerbosch.h"
#include "ramsey.h"
#include "clique_removal.h"
#include "mainindep.h"
#include <string.h>
#include <sys/resource.h>

int main(int argc, char *argv[]) {

TGraph *G;

char *fileName;
int graphType;
char algorithm[256];
int validAlgorithm;
char *printMode;

int *subSet;
int *independentSet;
```

```
struct rusage resourceUsage;

int i, j;

if (argc < 4) {
printf("Error: the comand line must contain:\n");
printf("* the input file name;\n");
printf("* the graph type. The availble types are:\n");
printf("  %d for DIRECTED graph;\n", DIRECTED);
printf("  %d for UNDIRECTED graph;\n", UNDIRECTED);
printf("* the print mode. The availble print modes are:\n");
printf("  %s for COMPACT PRINT mode;\n", COMPACT_PRINT_MODE);
printf("  %s for DETAILED PRINT mode);\n", DETAILED_PRINT_MODE );
printf("  %s for TP OUTPUT PRINT mode);\n", TP_OUTPUT_PRINT_MODE );
printf("* the algorithm to be used to solve the independence set
problem. The availble algorithms are:\n");
printf("  %s for the naive algoritm (optimum)\n", NAIVE_ALGORITHM);
printf("  %s for the Bron-Kerbosh algoritm (optimum)\n",
BRON_KERBOSCH_ALGORITHM);
printf("  %s for the Ramsey algorithm (approximated)\n",
RAMSEY_ALGORITHM);
printf("  %s for the Clique-Removal algorithm (approximated)\n",
CLIQUE_REMOVAL_ALGORITHM);
exit(-1);
} else if (argc < 5) {
printf("Warning: you have not defined the algorithm you want to use
to solve the independence set problem.\n");
printf("The availble algorithms are:\n");
printf("  %s for the naive algoritm (optimum)\n", NAIVE_ALGORITHM);
printf("  %s for the Bron-Kerbosh algoritm (optimum)\n",
BRON_KERBOSCH_ALGORITHM);
printf("  %s for the Ramsey algorithm (approximated)\n",
RAMSEY_ALGORITHM);
printf("  %s for the Clique-Removal algorithm (approximated)\n",
CLIQUE_REMOVAL_ALGORITHM);
printf("  %s for running ALL the alorithms above\n", ALL_ALGORITHMS);
printf("\n");
printf("The default algorithm to be used is the Bron-Kerbosch Algorithm
(optimum)\n");
strcpy(algorithm, BRON_KERBOSCH_ALGORITHM);
printf("\n");
} else {
strcpy(algorithm, argv[4]);
}

fileName = argv[1];
graphType = atoi(argv[2]);
printMode = argv[3];


validAlgorithm = FALSE;
```

```
G = readGraph(fileName, graphType);

if ( (strcmp(algorithm, NAIVE_ALGORITHM) == 0) || (strcmp(algorithm,
ALL_ALGORITHMS) == 0) ) {

validAlgorithm = TRUE;

if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
printf("\n");
printf("Using NAIVE algorithm\n");
printf("\n");
printf("Independent set size:\n");
}

independentSet = naive(G);

if ( (strcmp(printMode, COMPACT_PRINT_MODE) == 0) || (strcmp(printMode,
DETAILED_PRINT_MODE) == 0) )  {
printf("%i\n", vertexCount(independentSet, G->vertexNumber) );
}

if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
printf("Elements:\n");
}

printList(independentSet, G->vertexNumber);

}

if ( (strcmp(algorithm, BRON_KERBOSCH_ALGORITHM) == 0) || (strcmp(algorithm,
ALL_ALGORITHMS) == 0) ) {

validAlgorithm = TRUE;

if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
printf("\n");
printf("Using BRON-KERBOSCH algorithm\n");
printf("\n");
printf("Independent set size:\n");
}

// Modifica o grafo para o problema da clique
for (i=0; i<G->vertexNumber; i++) {
for (j=0; j<G->vertexNumber; j++) {
G->adjMatrix[i][j] = !G->adjMatrix[i][j];
}
}
for (i=0; i<G->vertexNumber; i++) {
G->adjMatrix[i][i] = TRUE;
}
//-------

independentSet = bron_kerbosch(G);
```

```
if ( (strcmp(printMode, COMPACT_PRINT_MODE) == 0) || (strcmp(printMode,
DETAILED_PRINT_MODE) == 0) )  {
printf("%i\n", vertexCount(independentSet, G->vertexNumber) );
}

if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
printf("Elements:\n");
}

printList(independentSet, G->vertexNumber);

// Modifica o grafo para o formato original
for (i=0; i<G->vertexNumber; i++) {
for (j=0; j<G->vertexNumber; j++) {
G->adjMatrix[i][j] = !G->adjMatrix[i][j];
}
}
for (i=0; i<G->vertexNumber; i++) {
G->adjMatrix[i][i] = FALSE;
}
//-------

}

if ( (strcmp(algorithm, RAMSEY_ALGORITHM) == 0) || (strcmp(algorithm,
ALL_ALGORITHMS) == 0) ) {

validAlgorithm = TRUE;


if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
printf("\n");
printf("Using RAMSEY algorithm\n");
printf("\n");
printf("Independent set size:\n");
}

subSet = (int*)malloc(G->vertexNumber * sizeof(int));
if (!subSet) {
printf("Error: insuficient memory for this operation! \n");
exit(-1);
}
setValues(subSet, G->vertexNumber, TRUE);

independentSet = ramsey(G, subSet)->I;

if ( (strcmp(printMode, COMPACT_PRINT_MODE) == 0) || (strcmp(printMode,
DETAILED_PRINT_MODE) == 0) )  {
printf("%i\n", vertexCount(independentSet, G->vertexNumber) );
}

if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
```

```
printf("Elements:\n");
}

printList(independentSet, G->vertexNumber);

free(subSet);

}

if ( (strcmp(algorithm, CLIQUE_REMOVAL_ALGORITHM) == 0) ||
(strcmp(algorithm, ALL_ALGORITHMS) == 0) ) {

validAlgorithm = TRUE;

if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
printf("\n");
printf("Using CLIQUE_REMOVAL algorithm\n");
printf("\n");
printf("Independent set size:\n");
}

subSet = (int*)malloc(G->vertexNumber * sizeof(int));
if (!subSet) {
printf("Error: insuficient memory for this operation! \n");
exit(-1);
}
setValues(subSet, G->vertexNumber, TRUE);

independentSet = cliqueRemoval(G, subSet);

if ( (strcmp(printMode, COMPACT_PRINT_MODE) == 0) ||
(strcmp(printMode, DETAILED_PRINT_MODE) == 0) )  {
printf("%i\n", vertexCount(independentSet, G->vertexNumber) );
}

if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
printf("Elements:\n");
}

printList(independentSet, G->vertexNumber);

free(subSet);

}

if (!validAlgorithm) {

printf("Error: the algorithm you want to use (code %s) is not
availble!\n", algorithm);
printf("The availble algorithms are:\n");
printf("  %s for the naive algoritm (optimum)\n", NAIVE_ALGORITHM);
printf("  %s for the Bron-Kerbosh algoritm (optimum)\n",
BRON_KERBOSCH_ALGORITHM);
```

```
printf("  %s for the Ramsey algorithm (approximated)\n",
RAMSEY_ALGORITHM);
printf("  %s for the Clique-Removal algorithm (approximated)\n",
CLIQUE_REMOVAL_ALGORITHM);
printf("  %s for running ALL the alorithms above\n", ALL_ALGORITHMS);
printf("\n");
exit(-1);

}

getrusage(RUSAGE_SELF, &resourceUsage);

if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
printf("\n");
printf("User time (us):\n");
}
if ( (strcmp(printMode, COMPACT_PRINT_MODE) == 0) || (strcmp(printMode,
DETAILED_PRINT_MODE) == 0) )  {
printf("%li\n", resourceUsage.ru_utime.tv_sec * 1000000 +
resourceUsage.ru_utime.tv_usec);
}

if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
printf("\n");
printf("System time (us):\n");
}
if ( (strcmp(printMode, COMPACT_PRINT_MODE) == 0) || (strcmp(printMode,
DETAILED_PRINT_MODE) == 0) )  {
printf("%li\n", resourceUsage.ru_stime.tv_sec * 1000000 +
resourceUsage.ru_stime.tv_usec);
}

if (strcmp(printMode, DETAILED_PRINT_MODE) == 0) {
printf("\n");
printf("Thank you for running my program! :-)\n");
}

return(0);
}
```

A.2.3 graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define TRUE 1
#define FALSE 0
#define DIRECTED 1
#define UNDIRECTED 0
```

```
typedef struct {

int **adjMatrix;
int **edgeList;
int vertexNumber;
int edgeNumber;
int type;

} TGraph;

TGraph *readGraph(char *fileName, int graphType);
void printGraph(TGraph *G);
int isIndependentSet(TGraph *G, int *set);
//int subSetGenerator(TGraph *G, int *subSet, int n, int k);
//void printSubSet(int *subSet, int size);
//void printList(int *subSet, int size);

#endif
```

A.2.4 graph.c

```
#include "graph.h"

TGraph *readGraph(char *fileName, int graphType){

TGraph *G;
FILE *fp;
int i, j, k;
int lim;

fp = fopen(fileName, "r");

if (!fp) {
printf("Error: Input file %s not found.\n", fileName);
exit(-1);
}

// Aloca o grafo G.
G = (TGraph*)malloc(sizeof(TGraph));

G->type = graphType;

// Lê o número de vértices de G.
fscanf(fp, "%i", &(G->vertexNumber));


// Aloca a matrix de adjacências de G.
G->adjMatrix = (int**)malloc(G->vertexNumber * sizeof(int*));

if (!G->adjMatrix) {
printf("Error: insuficient memory for this operation!");
exit(-1);
```

```
}

for(i=0; i<G->vertexNumber; i++){

G->adjMatrix[i] = (int*)malloc(G->vertexNumber * sizeof(int));

if (!G->adjMatrix[i]) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}

}

// Inicializa as arestas como todas inexistentes.
for (i=0; i<G->vertexNumber; i++) {
for (j=0; j<G->vertexNumber; j++) {
G->adjMatrix[i][j] = FALSE;
}
}

// Lê as arestas de G, preenchendo a matriz de adjacências.
while (!feof(fp)) {
fscanf(fp, "%d %d", &i, &j);
G->adjMatrix[i][j] = TRUE;

if (G->type == UNDIRECTED) {
G->adjMatrix[j][i] = TRUE;
}

}

// O número de arestas começa com 0;
G->edgeNumber = 0;

// Conta as arestas de G.
for (i=0; i<G->vertexNumber; i++) {

// Se o grafo é não-direcionado, só leva em conta a diagonal
superior da matriz de adjacências.
if (G->type == UNDIRECTED) {
lim = i;
} else {
// Se o grafo é direcionado, leva em conta a matriz de
adjacências inteira.
lim = 0;
}

for (j=lim; j<G->vertexNumber; j++) {
if (G->adjMatrix[i][j] == TRUE) {
G->edgeNumber++;
}
}
}
```

```
// Aloca a lista de arestas de G.
G->edgeList = (int**)malloc(G->edgeNumber * sizeof(int*));

if (!G->edgeList) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}

for(i=0; i<G->edgeNumber; i++){

G->edgeList[i] = (int*)malloc(2 * sizeof(int));

if (!G->edgeList[i]) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}

}

// Preenche a lista de arestas.
k = 0;

for (i=0; i<G->vertexNumber; i++) {

// Se o grafo é não-direcionado, só leva em conta a diagonal superior
da matriz de adjacências.
if (G->type == UNDIRECTED) {
lim = i;
} else {
// Se o grafo é direcionado, leva em conta a matriz de adjacências
inteira.
lim = 0;
}

for (j=lim; j<G->vertexNumber; j++) {
if (G->adjMatrix[i][j] == TRUE) {
G->edgeList[k][1] = i;
G->edgeList[k][2] = j;
k++;
}
}
}

fclose(fp);

return(G);

}

void printGraph(TGraph *G) {

int i, j, k;
```

```
// Imprime dados do grafo.
printf("Graph data:\n");
printf("Number of vertexes: %i\n", G->vertexNumber);
printf("Number of edges: %i\n", G->edgeNumber);
printf("Graph type (%i for UNDIRECTED, %i for DIRECTED): %i\n",
UNDIRECTED, DIRECTED, G->type);

printf("\n");

// Imprime a matriz de adjacências.
printf("Adjacency matrix:\n");
for (i=0; i<G->vertexNumber; i++) {
for (j=0; j<G->vertexNumber; j++) {
printf("%i ", G->adjMatrix[i][j]);
}
printf("\n");
}

printf("\n");

// Imprime a lista de arestas.
printf("Edge list:\n");
for (k=0; k<G->edgeNumber; k++) {
printf("(%i,%i)\n", G->edgeList[k][1], G->edgeList[k][2]);
}

}

int isIndependentSet(TGraph *G, int *set) {

int independent;
int k;
int v1, v2;

independent = TRUE;

for (k=0; k<G->edgeNumber; k++) {
v1 = G->edgeList[k][1];
v2 = G->edgeList[k][2];
if (set[v1] && set[v2]) {
independent = FALSE;
break;
}
}

return(independent);

}
```

A.2.5 module.h

```
#ifndef MODULE_H
```

```
#define define MODULE_H

#include <stdlib.h>
#include <stdio.h>

#define TRUE 1
#define FALSE 0

void printList(int *set, int size);
void printSet(int *set, int size);
int vertexCount(int *vertexList, int size);
void setValues(int *vertexList, int size, int value);
void listCopy(int *sourceList, int *targetList, int size);
void listSort(int *list, int minLim, int maxLim);

#endif
```

A.2.6 module.c

```
#include "module.h"

void printList(int *set, int size) {

int i;

for(i=0; i<size; i++) {
if (set[i] == TRUE) {
printf("%i ", i);
}
}
printf("\n");

}

void printSet(int *set, int size) {

int i;

printf("{ ");
for(i=0; i<size;i++) {
if (set[i] == TRUE) {
printf("%i ", i);
}
}
printf("}");
printf("\n");

}

int vertexCount(int *vertexList, int size) {

int count;
int i;
```

```
count = 0;
for (i=0; i<size; i++) {
if (vertexList[i] == TRUE) {
count++;
}
}

return(count);

}

void setValues(int *vertexList, int size, int value) {

int i;

for(i=0; i<size; i++) {
vertexList[i] = value;
}

}

void listCopy(int *sourceList, int *targetList, int size) {

int i;
for (i=0; i<size; i++) {
targetList[i] = sourceList[i];
}

}

void listSort(int *list, int minLim, int maxLim){

int i, j;
int aux;

for(i=minLim; i<maxLim-1; i++){
for(j=i; j<maxLim; j++){
if (list[j] < list[i]) {
aux = list[i];
list[i] = list[j];
list[j] = aux;
}
}
}

}
```

A.2.7 naive.h

```
#ifndef NAIVE_H
#define NAIVE_H
```

```
int *naive(TGraph *G);
int subSetGenerator(TGraph *G, int *subSet, int n, int k);

#endif
```

A.2.8 naive.c

```
#include "module.h"
#include "graph.h"
#include "naive.h"

int subSetGenerator(TGraph *G, int *subSet, int n, int k) {

int success;
int i;

success = FALSE;

if ( (k<=n) &&(k>=0) ) {

if (k==0) {

for (i=G->vertexNumber-n; i<G->vertexNumber; i++) {
subSet[i] = 0;
}

success = isIndependentSet(G, subSet);

} else {
subSet[G->vertexNumber-n] = 1;
success = subSetGenerator(G, subSet, n-1, k-1);
if (!success) {
subSet[G->vertexNumber-n] = 0;
success = subSetGenerator(G, subSet, n-1, k);
}
}

}

return(success);

}

int *naive(TGraph *G) {

int *subSet;
int isIndependent;
int k;
int supLimit;

subSet = (int*)malloc(G->vertexNumber * sizeof(int));
if (!subSet) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}

supLimit = (int)floor( (1 + sqrt(1 + 4*G->vertexNumber*G->vertexNumber
- 4*G->vertexNumber - 8*G->edgeNumber) )/2 );
```

```
isIndependent = FALSE;
for (k=supLimit; k>1; k--) {
isIndependent = subSetGenerator(G, subSet, G->vertexNumber, k);
if (isIndependent) {
break;
}
}

return(subSet);

}
```

A.2.9 bron-kerbosch.h

```
#ifndef BRON_KERBOSCH_H
#define BRON_KERBOSCH_H

typedef struct {

int *maxC;
int maxSize;

int *ALL;
int *compsub;
int c;

} TBK;

int connected(TGraph *G, int v1, int v2);
void extend_version2(int *old, int ne, int ce, TBK *bk, TGraph *G);
int  *bron_kerbosch(TGraph *G);

#endif
```

A.2.10 bron-kerbosch.c

```
#include "graph.h"
#include "module.h"
#include "bron_kerbosch.h"

int connected(TGraph *G, int v1, int v2){

return(G->adjMatrix[v1-1][v2-1] == TRUE);

}

void extend_version2(int *old, int ne, int ce, TBK *bk, TGraph *G) {

int *New;
int nod;
int fixp;
int newne;
```

```
int newce;
int i, j;
int count;
int pos;
int p;
int s;
int sel;
int minnod;

//int loc;

New = (int*)malloc( (ce+1) * sizeof(int) );

minnod = ce;
i = 0;
nod = 0;

// Determine each counter value and look for minimum.
for (i++; (i<=ce) && (minnod!=0); i++) {

p = old[i];
count = 0;
j = ne;

// Count disconections;
for (j++; (j<=ce) && (count<minnod); j++) {
if (!connected(G, p, old[j])) {
count++;
//Save position for potential candidate.
pos=j;
}
}

// Test for minimum.
if (count<minnod) {
fixp = p;
minnod = count;
if (i<=ne) {
s = pos;
} else {
s = i;
// Pre-increment.
nod = 1;
}
}

}

// If fixed point initially chosen from candidates, then number of
disconections will be pre-increased by one;


// ------------------------------------------------
```

```
// Backcycle
// ------------------------------------------------

for (nod += minnod; nod>=1; nod--) {

// Interchange;
p = old[s];
old[s] = old[ne+1];
sel = p;
old[ne+1] = p;

// Fill new set "not".
newne = 0;
i = 0;
for (i++; i<=ne; i++) {
if (connected(G, sel, old[i])) {
newne++;
New[newne] = old[i];
}
}

// Fill new set "cand"
newce = newne;
i = ne+1;
for (i++; i<=ce; i++) {
if (connected(G, sel, old[i])) {
newce++;
New[newce] = old[i];
}
}

// Add to "compsub"
(bk->c)++;
bk->compsub[bk->c] = sel;
if (newce == 0) {

/*
printf("clique = ");
for (loc=1; loc <=c; loc++) {
printf("%i ", compsub[loc]-1);
}
printf("\n");
*/

if (bk->c>bk->maxSize) {
listCopy(bk->compsub, bk->maxC, G->vertexNumber);
bk->maxSize = bk->c;
}


} else if (newne < newce) {
extend_version2(New, newne, newce, bk, G);
}
```

```
// "Remove from compsub"
(bk->c)--;

// Add to "not"
ne ++;
if (nod>1) {
// Select a candidate disconected to the fixed point
s = ne;
// Look for candidate;
LOOK: s++;
if (connected(G, fixp, old[s]) ){
goto LOOK;
}

}

}

// ------------------------------------------------
// End Backcycle
// ------------------------------------------------

free(New);

}

int *bron_kerbosch(TGraph *G) {

TBK *bk;
int *solution;
int i;

bk = (TBK*)malloc(sizeof(TBK));
if (!bk) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}

bk->ALL = (int*)malloc( (G->vertexNumber+1) * sizeof(int) );
if (!bk->ALL) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}

bk->compsub = (int*)malloc( (G->vertexNumber+1) * sizeof(int) );
if (!bk->compsub) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}

for(bk->c=1; bk->c <= G->vertexNumber; bk->c++) {
bk->ALL[bk->c] = bk->c;
```

```
}
bk->c = 0;

bk->maxC = (int*)malloc(G->vertexNumber * sizeof(int));
if (!bk->maxC) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}
bk->maxSize = 0;

extend_version2(bk->ALL,0,G->vertexNumber, bk, G);

//-------
solution = (int*)malloc( (G->vertexNumber) * sizeof(int) );
if (!solution) {
printf("Error: insuficient memory for this operation!");
exit(-1);
}

setValues(solution, G->vertexNumber, FALSE);

for (i=1; i<=bk->maxSize; i++) {
solution[bk->maxC[i]-1] = TRUE;
}
//-------

free(bk->ALL);
free(bk->compsub);
free(bk->maxC);
free(bk);

return(solution);

}
```

A.2.11 ramsey.h

```
#ifndef RAMSEY_H
#define RAMSEY_H

typedef struct{

int *C;
int *I;

} TAnswer;

void printAnswer(TAnswer *answer, int size);
int chooseVertex(int *vertexList, int size);
void fillWithNeighboors(int *vertexList, int *subGraph, TGraph *G, int v);
void fillWithNonNeighboors(int *vertexList, int *subGraph, TGraph *G, int v);
TAnswer *ramsey(TGraph *G, int *subGraph);
```

```
#endif
```

A.2.12 ramsey.c

```
#include "module.h"
#include "graph.h"
#include "ramsey.h"

void printAnswer(TAnswer *answer, int size) {

printf("Clique:\n");
printSet(answer->C, size);

printf("Independent set:\n");
printSet(answer->I, size);

}

int chooseVertex(int *vertexList, int size) {

int v;
int i;

for (i=0; i<size; i++) {
if (vertexList[i] == TRUE) {
v = i;
break;
}
}

return(v);

}

void fillWithNeighboors(int *vertexList, int *subGraph, TGraph *G, int v) {

int i;

for(i=0; i<G->vertexNumber; i++) {

vertexList[i] = subGraph[i];

if(!G->adjMatrix[i][v]) {
vertexList[i] = FALSE;
}

}

vertexList[v] = FALSE;

}

void fillWithNonNeighboors(int *vertexList, int *subGraph, TGraph *G, int v) {
```

```
int i;

for(i=0; i<G->vertexNumber; i++) {

vertexList[i] = subGraph[i];

if(G->adjMatrix[i][v]) {
vertexList[i] = FALSE;
}

}

vertexList[v] = FALSE;

}

TAnswer *ramsey(TGraph *G, int *subGraph) {

TAnswer *answer, *subAnswer1, *subAnswer2;
int *N1, *N2;
int v;

answer = (TAnswer*)malloc(sizeof(TAnswer));
if (!answer) {
printf("Error: insuficient memory for this operation! \n");
exit(-1);
}

answer->C = (int*)malloc(G->vertexNumber * sizeof(int));
if (!answer->C) {
printf("Error: insuficient memory for this operation! \n");
exit(-1);
}

answer->I = (int*)malloc(G->vertexNumber * sizeof(int));
if (!answer->I) {
printf("Error: insuficient memory for this operation! \n");
exit(-1);
}

if (vertexCount(subGraph, G->vertexNumber) == 0) {
setValues(answer->C, G->vertexNumber, FALSE);
setValues(answer->I, G->vertexNumber, FALSE);
return(answer);
}

v = chooseVertex(subGraph, G->vertexNumber);

N1 = (int*)malloc(G->vertexNumber * sizeof(int));
if (!N1) {
printf("Error: insuficient memory for this operation! \n");
exit(-1);
```

```
}
fillWithNeighboors(N1, subGraph, G, v);

N2 = (int*)malloc(G->vertexNumber * sizeof(int));
if (!N1) {
printf("Error: insuficient memory for this operation! \n");
exit(-1);
}
fillWithNonNeighboors(N2, subGraph, G, v);

subAnswer1 = ramsey(G, N1);
subAnswer2 = ramsey(G, N2);

subAnswer1->C[v] = TRUE;
subAnswer2->I[v] = TRUE;

if (vertexCount(subAnswer1->C, G->vertexNumber) > vertexCount(subAnswer2->C, G->vertexNumber)) {
listCopy(subAnswer1->C, answer->C, G->vertexNumber);
} else {
listCopy(subAnswer2->C, answer->C, G->vertexNumber);
}

if (vertexCount(subAnswer1->I, G->vertexNumber) > vertexCount(subAnswer2->I, G->vertexNumber)) {
listCopy(subAnswer1->I, answer->I, G->vertexNumber);
} else {
listCopy(subAnswer2->I, answer->I, G->vertexNumber);
}

//TAnswerFree(subAnswer1);
free(subAnswer1->C);
free(subAnswer1->I);
free(subAnswer1);

//TAnswerFree(subAnswer1);
free(subAnswer2->C);
free(subAnswer2->I);
free(subAnswer2);

free(N1);
free(N2);

return(answer);

}
```

A.2.13 clique-removal.h

```
#ifndef CLIQUE_REMOVAL_H
#define CLIQUE_REMOVAL_H

void removeSubGraph(int *graph, int *subGraph, int size);
int *cliqueRemoval(TGraph *G, int *subGraph);
```

```
#endif
```

A.2.14 clique-removal.c

```
#include "module.h"
#include "graph.h"
#include "ramsey.h"
#include "clique_removal.h"

void removeSubGraph(int *graph, int *subGraph, int size) {

int i;

for (i=0; i<size; i++) {
if (graph[i] && subGraph[i]) {
graph[i] = FALSE;
}
}

}

int *cliqueRemoval(TGraph *G, int *subGraph) {

TAnswer *answer;
int *maxI;
int maxSize;
int *newSubGraph;

maxI = (int*)malloc(G->vertexNumber * sizeof(int));
if (!maxI) {
printf("Error: insuficient memory for this operation! \n");
exit(-1);
}

newSubGraph = (int*)malloc(G->vertexNumber * sizeof(int));
if (!newSubGraph) {
printf("Error: insuficient memory for this operation! \n");
exit(-1);
}
listCopy(subGraph, newSubGraph, G->vertexNumber);

answer = ramsey(G, subGraph);
listCopy(answer->I, maxI, G->vertexNumber);
maxSize = vertexCount(maxI, G->vertexNumber);

while (vertexCount(newSubGraph, G->vertexNumber) != 0) {

removeSubGraph(newSubGraph, answer->C, G->vertexNumber);

free(answer);
answer = ramsey(G, newSubGraph);

if (vertexCount(answer->I, G->vertexNumber) > maxSize) {
```

```
listCopy(answer->I, maxI, G->vertexNumber);
maxSize = vertexCount(maxI, G->vertexNumber);
}

}

free(newSubGraph);

return(maxI);

}
```

A.3 makefile

```
# PAA - Exemplo de um arquivo makefile.
# Precisamos de um arquivo chamado makefile para dizer ao aplicativo make o que fazer.
# Geralmente, o makefile diz ao make como compilar um programa.
# 1. Instrucoes para construcao do makefile:
#    a. Nesse exemplo, "maindisted.c" e o programa principal para o primeiro problema,
#       "mainindep.c" e o programa principal para o segundo problema e "module.c" e um
#       modulo que cotem funcoes usadas pelos progrmas principais
#    b. Antes dos comandos "./disted", "./indep", "/usr/bin/$(CC)" e "rm" deve existir
#       um caractere de tabulacao (tecla "tab").
# 2. Instrucoes para execucao:
#    a. Para compilar e executar o seu codigo, gerando o arquivo saidadisted.tp2,
#       execute o comando "make rundisted" no prompt de comandos.
#    b. Para compilar e executar o seu codigo, gerando o arquivo saidaindep.tp2,
#       execute o comando "make runindep" no prompt de comandos.
#    c. Para apagar os arquivos executaveis e objetos, execute o comando
#       "make clean" no prompt de comandos.

CC = g++ -Wall
objects = module.o graph.o
algorithms = naive.o bron_kerbosch.o ramsey.o clique_removal.o
graphType = 0
printMode = D
algorithmToBeUsed = ALL

all : compdisted compindep

module.o: module.c module.h
/usr/bin/$(CC) -c module.c -o module.o

graph.o: graph.c graph.h
/usr/bin/$(CC) -c graph.c -o graph.o

naive.o: naive.c naive.h
/usr/bin/$(CC) -c naive.c -o naive.o

bron_kerbosch.o: bron_kerbosch.c bron_kerbosch.h
/usr/bin/$(CC) -c bron_kerbosch.c -o bron_kerbosch.o

ramsey.o: ramsey.c ramsey.h
```

```
/usr/bin/$(CC) -c ramsey.c -o ramsey.o

clique_removal.o: clique_removal.c clique_removal.h
/usr/bin/$(CC) -c clique_removal.c -o clique_removal.o

compdisted: maindisted.c maindisted.h $(objects)
/usr/bin/$(CC) maindisted.c $(objects) -o disted

rundisted: compdisted
./disted entradadisted.tp2 > saidadisted.tp2

compindep: mainindep.c mainindep.h $(objects) $(algorithms)
/usr/bin/$(CC) mainindep.c $(objects) $(algorithms) -o indep

runindep: compindep
./indep entradaindep.tp2 $(graphType) $(printMode) $(algorithmToBeUsed)

clean :
rm -f *.o rundisted runindep
```

Referências

[1] Dan Gusfield. Algorithms on strings, trees and sequences. *Cambridge University Press*, 1997

[2] C. Bron and Joep Kerbosch. Finding all cliques on an undirected graph. *Communications of the ACM*, Vol.16, Número 9, Setembro de 1973

[3] N. Christofides. Graph theory: an algorithm approach, *Academic Press*, capítulo 3, págs. 30-35, 1975

[4] R. Boppana and M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. In *SWAT*, pages 11-25. Springer Verlag, 1990.

[5] S. Homer and M. Peinado. On the performance of polynomial-time clique approximation algorithms on very large graphs. *Cliques, Coloring, and Satisfiability: second DIMACS Implementation Challenge*, volume 26, páginas 103-104. DIMACS American Mathematical Society, 1996

[6] V. T. Paschos. A survey of approximately optimal solutions to some covering and packing problems *ACM Computing Surveys*, Vol. 29, No. 2, Junho de 1997