UNIVERSIDADE FEDERAL DO CEARÁ – UFC

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

MARCIO COSTA SANTOS

EXPERIMENTOS COMPUTACIONAIS COM IMPLEMENTAÇÕES DE CONJUNTOS POR ENDEREÇAMENTO DIRETO E O PROBLEMA DE CONJUNTO INDEPENDENTE MÁXIMO

Fortaleza - Ceará, Brasil

13 DE SETEMBRO DE 2013

MARCIO COSTA SANTOS

EXPERIMENTOS COMPUTACIONAIS COM IMPLEMENTAÇÕES DE CONJUNTOS POR ENDEREÇAMENTO DIRETO E O PROBLEMA DE CONJUNTO INDEPENDENTE MÁXIMO

Dissertação de Mestrado apresentada ao Programa de Mestrado e Doutorado em Ciência da Computação, do Departamento de Computação da Universidade Federal do Ceará, como requisito para obtenção do Título de Mestre em Ciência da Computação. Área de concentração: Otimização (Teoria da Computação)

UNIVERSIDADE FEDERAL DO CEARÁ – UFC

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

Orientador: Prof. Dr. Ricardo Cordeiro Corrêa

Fortaleza - Ceará, Brasil

13 de Setembro de 2013

MARCIO COSTA SANTOS
EXPERIMENTOS COMPUTACIONAIS COM IMPLEMENTAÇÕES DE CONJUNTOS POR ENDEREÇAMENTO DIRETO E O PROBLEMA DE CONJUNTO INDEPENDENTE MÁXIMO/ MARCIO COSTA SANTOS. – Fortaleza - Ceará, Brasil, 13 de Setembro de 2013-
78 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Ricardo Cordeiro Corrêa

Dissertação de Mestrado – UNIVERSIDADE FEDERAL DO CEARÁ – UFC
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO, 13 de Setembro de 2013.

1. Mapas de Bits. 2. Conjunto Independente Máximo. 3. Paralelismo de Bits I. Prof. Dr. Ricardo Cordeiro Corrêa. II. Universidade Federal do Ceará. III. Departamento de Computação.

CDU 02:141:005.7

AGRADECIMENTOS

Gostaria de agradecer primeiramente a deus, pois sem ele nada é possível, e a toda minha família pelo apoio durante este trabalho, especialmente a minha mãe.

Depois gostaria de agradecer a todos os membros da banca, que se dispuseram a ler o trabalho com um prazo diminuto de tempo e em especial ao meu orientador professor Ricardo Corrêa.

Agradeço também a todos os meus amigos do *lab2*, Rafael, Rennan, Eliezer, Vinícius, Arthur e Paulo Henrique que passaram por muitos momentos divertidos e muitas disciplinas problemáticas sempre tentando provar algum resultado e dominar o mundo nas horas vagas.

Agradeço ainda as minhas amigas Lívia, Ticiana, Danusa e Karol pelo apoio nestes quase 7 anos de UFC e pelo incentivo para a conclusão deste trabalho. Em especial, a Camila Nair por sua paciência e compreensão e, em momentos necessários, a falta da mesma.

Por fim gostaria de agradecer a todos os professores do *Pargo* que contribuíram, até bem demais, para a minha formação profissional, em especial para o professores Victor Campos e Ana Shirley.

Em especial, gostaria de agradecer a 3 pessoas que contribuíram muito negativamente para este trabalho, mas muito positivamente para a minha formação, não só profissional. São eles, meus amigos Phablo Moura e Tatiane Figueredo, cujo maior objetivo acredito seja conhecer todo o mundo, e o professor Manoel Câmpelo, que apesar das disciplinas dificílimas e das listas de exercícios *infazíveis*, sempre esteve muito presente e nestes 5 anos de *Pargo*, muitas vezes, foi bem além de seus deveres para ajudar não só a mim, mas muitos outros alunos do grupo.

RESUMO

A utilização de vetores de bits é prática corrente na representação de conjuntos por endereçamento direto com o intuito de reduzir o espaço de memória necessário e melhorar o desempenho de aplicações com uso de técnicas de paralelismo em bits.

Nesta dissertação, examinamos implementações para representação de conjuntos por endereçamento direto. A estrutura básica nessas implementações é o vetor de bits. No entanto, além dessa estrutura básica, implementamos também duas variações. A primeira delas consiste em uma estratificação de vetores de bits, enquanto a segunda emprega uma tabela de dispersão.

As operações associadas às estruturas implementadas são a inclusão ou remoção de um elemento do conjunto e a união ou interseção de dois conjuntos. Especial atenção é dada ao uso de paralelismo em bits nessas operações. As implementações das diferentes estruturas nesta dissertação utilizam uma interface e uma implementação abstrata comuns, nas quais as operações são especificadas e o paralelismo em bits é explorado. A diferença entre as implementações está apenas na estrutura utilizada. Uma comparação experimental é realizada entre as diferentes estruturas utilizando algoritmos enumerativos para o problema de conjunto independente máximo.

Duas abordagens são utilizadas na implementação de algoritmos enumerativos para o problema de conjunto independente máximo, ambas explorando o potencial de paralelismo em bits na representação do grafo e na operação sobre subconjuntos de vértices. A primeira delas é um algoritmo do tipo *branch-and-boound* proposto na literatura e a segunda emprega o método das bonecas russas. Em ambos os casos, o uso de paralelismo em bits proporciona ganhos de eficiência quando empregado no cálculo de limites inferiores baseados em cobertura por cliques. Resultados de experimentos computacionais são apresentados como forma de comparação entre os dois algoritmos e como forma de avaliação das estruturas implementadas. Esses resultados permitem concluir que o algoritmo baseado no método das bonecas russas é mais eficiente quanto ao tempo de execução e quanto ao consumo de memória. Além disso, os resultados experimentais mostram também que o uso de estratificação e tabelas de dispersão permitem ainda maior eficiência no caso de grafos com muito vértices e poucas arestas.

Palavras-chaves: Conjunto independente em grafos. Método de *branch-and-bound*. Método das Bonecas Russas. Endereçamento direto e paralelismo

de bits.

ABSTRACT

The use of bit vectors is a usual practice for represent sets by direct addressing with the aim of reduce memory consumed and improve efficiency of applications with the use of bit parallel techniques.

In this text, we study implementations for represent sets by direct addressed. The basic structure in this implementations is the bit vector. Besides that basic implementation, we implement two variations also. The first one is a stratification of the bit vector, while the second uses a hash table.

The operations linked to the implemented structure are include and remove an element and the union and intersection of two sets. Especial attention is given to the use of bit parallel in this condition. The implementation of the different structures in this work use an base interface and a base abstract class, where the operations are defined and the bit parallel is used. An experimental comparative between this structures is carry out using enumerative algorithms for the maximum stable set problem.

Two approaches are used in the implementation of the enumerative algorithms for the maximum stable set problem, both using the bit parallel in the representation of the graph and on the operations with subsets of vertices. The first one is a known branch-and-bound algorithm and the second uses the Russian dolls method. In both cases, the use of bit parallel improve efficiency when the lower bounds are calculated based in a clique cover of the vertices. The results of computational experiments are presented as comparison between the two algorithms and as an assessment of the structures implemented. These results show that the algorithm based on the method Russian Dolls is more efficient regarding runtime and the memory consumed. Furthermore, the experimental results also show that the use stratification and hash tables also allow more efficiency in the case of sparse graphs.

Key-words: Stable set of graphs. Direct addressed and bit parallel. Russian dolls method. Branch-and-bound.

SUMÁRIO

1 INTRODUÇÃO

A técnica de *armazenamento por endereçamento direto* é uma forma eficiente de representação de um conjunto A definido sobre um universo $U = \{0, 1, \ldots, u-1\}$, cujo tamanho u, além de conhecido de antemão, não é abusivamente grande. Uma forma de representar A é usar uma tabela T de tamanho u fazendo com que os elementos de U sejam índices em T. O conteúdo de $T[k]$, para todo $k \in A$, pode ser um valor associado ao elemento k ou simplesmente uma informação sobre a presença de k em A. Para os elementos de U não presentes em A, os conteúdos correspondentes em T são simplesmente uma informação da ausência desse elemento em A. Observe que os elementos de A são mantidos ordenados em T, segundo a ordem usual representada pelo operador $\leq$.

Nesta dissertação, o armazenamento por endereçamento direto é usado de forma que apenas a informação sobre a presença ou não de um elemento no conjunto é armazenada. Por essa razão, estudamos o que passamos a denominar de *mapa de bits*, que nada mais é do que a aplicação do principio do armazenamento por endereçamento direto com a ideia de usar um bit associado a cada elemento do universo. O valor desse bit é 1 quando o seu índice corresponde a um elemento do conjunto representado, e 0 em caso contrário. A forma mais direta de se implementar um mapa de bits é usando um vetor, o que chamamos de *vetor de bits*. Nesta dissertação, estudamos também algumas formas mais elaboradas de implementação de mapas de bits.

Nas seções a seguir, abordamos em mais detalhes esses aspectos. Para os detalhes de notação, o leitor pode se referir ao Apêndice A.

Na literatura temos poucos exemplos de trabalhos que tentam explorar diferentes implementações para mapas de bits e utilizar todos os conceitos apresentados anteriormente. A grande maioria utiliza as técnicas descritas acima apenas para vetores de bits. Nesta dissertação realizamos uma implementação versátil e eficiente de mapas de bits através de orientação a objetos, visando a utilização de todas as técnicas descritas anteriormente. Para testarmos estas implementa-

ções fazemos uso de algoritmos exatos para o problema de conjunto independente máximo que fazem uso intensivo de mapas de bits.

Este trabalho é parte das atividades do projeto de cooperação científica STAB no programa STIC/AmSud, financiado pela CAPES (Brasil), CNRS e MAE (França), CONICYT (Chile) e MINCYT (Argentina) e do projeto *Paralelismo, Grafos e Otimização* no programa PRONEM, financiado pela FUNCAP (estado do Ceará) e CNPQ (Brasil).

1.1 Implementações de Mapas de Bits

Como é natural em toda estrutura de dados para conjuntos, além da descrição da forma de representação dos elementos, descrevemos também algoritmos para a realização de algumas operações relevantes sobre conjuntos usando mapas de bits. Essas operações são a inclusão e a remoção de um elemento do conjunto, a união, a interseção, a diferença e a diferença simétrica entre dois conjuntos, além do complemento de um conjunto. Para tornar mais simples a exposição de propriedades de mapas de bits feita a seguir, consideramos a implementação baseada em um vetor de bits, embora essas propriedades também sejam válidas para outras formas de implementação estudadas nesta dissertação.

Um *nó* é um conjunto sucessivo de bits que pode ser armazenado internamente em um registrador. Uma hipótese verificada na maioria dos computadores atuais e, por conseguinte, aceita nesta dissertação é que um nó pode ser transferido entre a memória e o registrador em apenas uma operação de endereçamento de memória. Definimos o *tamanho* de um nó como sendo o número de bits que ele ocupa na memória. Este valor é denotado por w no decorrer do texto. Dois valores típicos para w são 32 e 64 bits.

Um vetor de bits representando um conjunto A de um universo U é uma tabela B_A de $\lceil \frac{u}{w} \rceil$ nós em que um nó $B_A[i]$ contém os bits referentes aos elementos $iw, iw+1, \ldots, (i+1)w-1$ do universo U. Usamos a notação usual da linguagem C para representar as operações lógicas de E, OU, OU exclusivo e negação, assim como as operações de deslocamento de bits à direita e à esquerda ($\&, |, \wedge, \sim, \ll, \gg$ respectivamente). A consulta a um elemento $e \in U$ é feita em duas dimensões. Para

endereçar o nó correspondente em B_A, calcula-se a função $nodeIdx(e) = \lfloor \frac{e}{w} \rfloor$. Além disso, o índice do bit em $B_A[nodeIdx(e)]$ é dado pela função $idxInNode(e) = e \bmod w$. Empregando essas funções, o resultado da operação

$$B_A[nodeIdx(e)]\&(1LL \ll idxInNode(e))$$

produz um nó cujo valor é diferente de 0 se e somente se $e \in A$. De maneira análoga as operações

$$B_A[nodeIdx(e)] = B_A[nodeIdx(e)] \mid (1LL \ll idxInNode(e))$$

e

$$B_A[nodeIdx(e)] = B_A[nodeIdx(e)]\& \sim (1LL \ll idxInNode(e))$$

representam a inclusão e remoção, respectivamente, do elemento e no conjunto A.

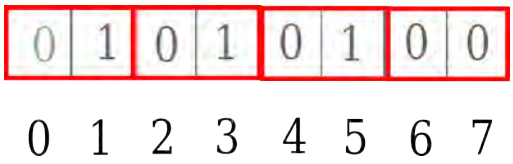


Figura 1: Vetor de bits representando o conjunto $\{1, 3, 5\}$ do universo $\{0, 1, 2, 3, 4, 5, 6, 7\}$ com tamanho do nó $w = 2$.

As operações $\lfloor \frac{e}{w} \rfloor$ e $e \bmod w$ podem ser implementadas apenas com deslocamento de bits se u e w são potências de 2 e o valor de $\log w$ é conhecido previamente. Suponha que $w = 2^p$, $u = 2^q$ e $\log w = p$. A operação $\lfloor \frac{e}{w} \rfloor$ corresponde a dividir um número em base binária, por uma potência de 2, o que corresponde a deslocar $\log w$ bits de e para a direita. Logo $\lfloor \frac{e}{w} \rfloor e \gg \log w$. Por outro lado, a operação $e \bmod w$ consiste em considerar apenas os $\log w$ menos significativos da palavra, $e \,\&\, mask$, onde $mask$ é um nó onde todos os bits de posição maior que $\log w$ são zeros e todos os menores são 1.

Uma vez que os mapas de bits representam conjuntos como agrupamentos de bits, operações lógicas podem ser utilizadas para efetuar as operações sobre conjuntos. Dados dois conjuntos A e A' e suas representações como vetores de bits B_A e $B_{A'}$, podemos traduzir as operações sobre conjuntos como descrevemos a seguir:

$A \cup A' \equiv B_A \mid B_{A'}$

$A \cap A' \equiv B_A \ \& \ B_{A'}$

$\bar{A} \equiv \sim B_A$

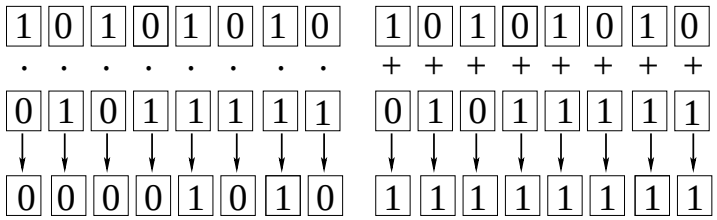
$A \setminus A' \equiv B_A \ \& \sim B_{A'}$

$A \oplus A' \equiv B_A \wedge B_{A'}$

1.2 Paralelismo em Bits

Paralelismo em bits é uma técnica usada para efetuar as operações sobre mapas de bits capacidade que os microprocessadores têm de realizar operações lógicas, bit por bit, simultaneamente sobre todos os bits armazenados em seus registradores internos. Esta técnica começou a ser estudada no final da década de 80 e vem sendo aplicada com sucesso em vários problemas em grafos (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011).

Uma operação lógica, bit por bit, sobre dois nós N_1 e N_2 significa que essa operação é realizada simultaneamente tomando-se como operandos dois bits correspondentes de N_1 e N_2. Por exemplo, sejam $N_1 = 10101010$ e $N_2 = 01011111$ duas palavras de tamanho 8 e a operação lógica E. A notação, a operação N_1 & N_2 produz como resultado a palavra 0001010, palavra esta que é obtida pela realização simultânea das operações indicadas na Figura 2. Ainda nessa figura, temos também o exemplo da operação lógica OU com as mesmas palavras N_1 e N_2.



(a) Operação E (b) Operação OU

Figura 2: Exemplo de realizações das operações lógicas E e OU empregando paralelismo em bits.

O potencial ganho de desempenho oriundo do paralelismo presente nas operações entre conjuntos representados por mapas de bits é evidente. Uma implementação cuidadosa pode explorar esse potencial conforme mostrado em (BAEZA-YATES; GONNET, 1992). O cenário torna-se um pouco mais exigente quando a computação sobre os conjuntos envolve a enumeração de seus elementos. Neste caso, é preciso lançar mão de uma operação de determinação do bit de valor 1 de menor índice em um nó, denominada $LSB(e)$ (do inglês *least significant bit* de um nó e). Em muitos casos, tal operação está disponível na arquitetura do hardware (Um exemplo, que utilizamos na implementação, são as *funções built-in*). Mesmo quando este não é o caso, soluções envolvendo tabelas de consulta (SEGUNDO; RODRIGUEZ-LOSADA; ROSSI, 2007) ou funções de dispersão baseada em sequências de *DeBruijn* (LEISERSON; PROKOP; RANDALL, 1998) são eficientes.

A operação $LSB(B_A)$ é a generalização da determinação do bit de menor índice para o vetor de bits B_A. A sua implementação se faz verificando consecutivamente cada nó $B_A[i]$ desde $i = 0$ até encontrar o índice de um nó não nulo de B_A, digamos j. Então, $jw + LSB(B_A[j])$ é o elemento procurado. A complexidade desta operação é, no pior caso, proporcional a $\lceil \frac{u}{w} \rceil$. A utilização de $LSB(e)$ para a enumeração dos elementos em B_A implica a sua aplicação para cada um desses elementos, gerando um acréscimo de tempo proporcional a $|A|$, o que pode tornar o uso de vetores de bits ineficientes face ao uso de tabelas de inteiros (SEGUNDO; RODRIGUEZ-LOSADA; ROSSI, 2007).

Por fim, observe que o uso de vetores de bits usualmente proporciona uma economia considerável de memória pois apenas 1 bit é associado a cada elemento do conjunto. Além desta economia, temos a vantagem da utilização do paralelismo de bits para realizar as operações sobre os conjuntos.

1.3 Enumeração e Conjuntos Esparsos

1.3.1 Estratificação

Ainda sobre a enumeração dos elementos de um mapas de bits, observe que a um conjunto esparso pode corresponder um mapa de bits possuindo um ou mais nós vazios (sem bits de valor 1). Durante a enumeração dos elementos de um conjunto, a utilização de um *sumário* permite evitar percorrer esses nós vazios. Um exemplo é ilustrado na Figura 2. Um *sumário* S para o mapa de bit T é um mapa de bits sobre o universo $\{1, \ldots, \lceil \frac{u}{w} \rceil\}$ com a seguinte semântica: $S[i] = 1$ se e, somente se, o i-ésimo nó de T não for nulo, ou seja, o valor de $S[i]$ é resultado do OU lógico de $T[iw], \ldots T[(i+1)w-1]$. Essa ideia pode ser aplicada novamente ao próprio sumário, uma vez que este é um mapa de bits também. Este processo é o que denominamos de *estratificação* de um mapa de bits, visto que produz uma estrutura que pode ter vários níveis.

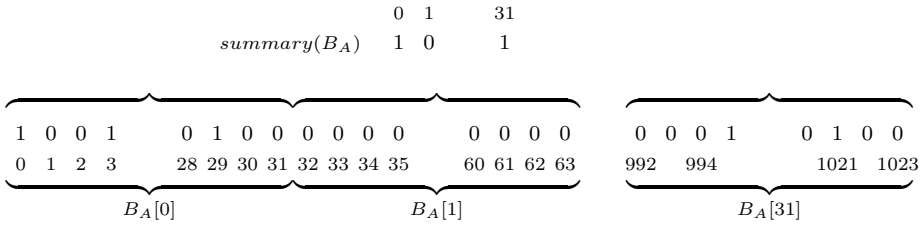


Figura 3: Representação gráfica de um sumário sobre um vetor de bits.

A inclusão destes sumários tem como principal objetivo acelerar a operação de enumeração dos elementos do conjunto, evitando a visita a nós vazios (sem bits de valor 1). Para enumerar os elementos do conjunto, recursivamente, buscamos no sumário o primeiro bit de valor 1, seja este j, e então procuramos no mapa de bits de base o primeiro bit 1 no nó j. A partir deste primeiro elemento, enumeramos os elementos no nó ao qual ele pertence e depois retornamos ao sumário para determinar o próximo nó não vazio do vetor de base.

Uma forma alternativa de descrever um mapa de bits estratificado é como uma árvore onde cada bit do sumário é pai dos bits presentes no nó que este representa. Esta forma de ver a estrutura com vários sumários sugere que esta

é uma simplificação da *árvore de van Emde Boas* (CORMEN; RIVEST; STEIN, 2003) de forma a usar percurso linear nos caminhos folha-raiz nas operações sobre a árvore. Embora teoricamente os percursos lineares sejam menos eficientes que os percursos binários das árvores de van Emde Boas originais, nós advogamos que o uso da nossa árvore adaptada é mais eficiente na prática como decorrência dos seguintes fatos:

- Operações sobre mapas de bits podem ser realizadas de maneira muito eficiente com instruções específicas do processador. Esta não é uma propriedade das árvores van Emde Boas quando se considera o tratamento diferenciado que o menor e o maior elemento de cada nível recebe (CORMEN; RIVEST; STEIN, 2003).

- A altura de uma árvore de van Emde Boas é pequena para todas as aplicações práticas quando se considera nós de $w = 32$ ou 64 bits. Considerando o tamanho do nó como 32, uma árvore de altura 2 com 32 nós no sumário representa um universo de $32.32.32 = 32768$ elementos. O tamanho do universo passa a ser 262144 se tomássemos o tamanho do nó como sendo $w = 64$, uma árvore de altura 2 e 64 elementos no sumário. Com isso, nos casos práticos, a altura 2 já é suficiente. Portanto, a diferença entre o percurso linear e o percurso binário nos caminhos folha-raiz é bem pequeno.

1.3.2 Tabelas de Dispersão

Ainda mantendo nossa atenção nos conjuntos esparsos, uma implementação alternativa para os mapas de bits é uma *tabela de dispersão* onde cada entrada da tabela é um nó.

Uma *tabela de dispersão* para um universo $U = \{0, \ldots, u\}$ é uma tabela de m posições, onde cada posição corresponde um nó não vazio da representação de U por meio de um vetor de bits B_U. A tabela de dispersão á ainda dotada da função $nodeidx : U \to 0, 1, \ldots, \lceil \frac{u}{w} \rceil$, indicando em que posição da tabela T está o nó correspondente a cada elemento do universo. Observe que a função $nodeidx(e)$ torna-se uma função de dispersão mais complexa que a função $\lfloor \frac{e}{w} \rfloor$ usada no caso

$m = \lceil \frac{u}{w} \rceil$. O emprego da tabela de dispersão na implementação de mapas de bits incorre em custos extras de tempo (para calcular a função de dispersão para todos os acessos aos elementos da tabela) e de armazenamento (para representar a função de dispersão). Estes custos são justificados pelo economia de memória proporcionada pela tabela de dispersão em conjuntos muito esparsos, já que não precisamos armazenar nós que são vazios. Essa econômica de memória permanecesse, e pode ser ainda maior, quando usamos a estratificação para reduzir o tempo de enumeração do conjunto representado pela tabela de dispersão.

1.4 Algoritmos Exatos para o Problema de Conjunto Independente Máximo em Grafos

Dado um grafo simples e não direcionado $G = (V, E)$ o *Conjunto Independente Máximo (CIM)* consiste em determinar $\alpha(G)$.

Este problema tem muitas aplicações em biologia computacional, otimização de compiladores e redes de telecomunicações dentre outras áreas {(PARDALOS; XUE, 1994), (BOMZE; BUDINICH; M., 1999), (BEN-DOR; SHAMIR; YAKHINI, 1999)}. Este problema também aparece como subproblema em diversos problemas de otimização combinatória, como o problema de coloração por exemplo.

O problema CIM é NP-Difícil para grafos arbitrários. Em (PAPADIMITRIOU; YANNAKAKIS, 1981), Papadimitrou e Yannakakis introduziram a classe de complexidade $MAXSNP$ e mostraram que muitos problemas, inclusive o CIM, são completos nesta classe. Sabe-se ainda que não existe algoritmo polinomial aproximativo para o problema CIM com um fator de $O(|V(G)|^{1-\epsilon})$, exceto se $P = NP$ (AURORA et al., 1992) e (AURORA; SAFRA, 1992).

Os algoritmos que produzem soluções exatas para o problema *CIM* podem ser classificados em duas classes distintas: Algoritmos Enumerativos e Algoritmos de Programação Linear Inteira. O mais estudado dos modelos de programação linear inteira é o que apresentamos a seguir. Define-se uma variável binária $x_v \in \{0, 1\}$ associada a cada $v \in V(G)$, assim

$$\alpha(G) = \max \sum_{v \in V(G)} x_v$$
$$\text{tal que}$$
$$x_v + x_u \leq 1 \text{ para todo } uv \in E(G)$$
$$x_v \in \{0, 1\}$$

Vários estudos poliédricos existem sobre esta formulação e muitos algoritmos de *branch-and-bound* e de *branch-and-cut* já foram utilizados para resolver este problema inteiro. Este tipo de abordagem é muito útil quando lidamos com o problema CIM na forma de um subproblema gerado pelas restrições, ou por um subconjunto das mesmas, de outro problema que utiliza programação inteira (PARDALOS; XUE, 1994) e (SMRIGLIO et al., 2008).

As implementações que tomamos como aplicação de mapas de bits nesta dissertação envolvem algoritmos enumerativos, uma vez que estes fazem uso mais intenso dos mapas de bits e de todo o ferramental do paralelismo de bits.

1.5 Algoritmos Enumerativos para o Problema CIM

Antes de começarmos a apresentar os algoritmos enumerativos devemos fazer uma observação. Apresentamos versões recursivas de todos os algoritmos que estudamos, mas implementamos versões iterativas dos mesmos. Isso ocorre porque a recursão consume mais tempo de processamento e memória para ser realizada. Optamos por apresentar as versões recursivas na descrição dos algoritmos porque são mais intuitivas e facilitam a exposição. Além disso, qualquer algoritmo recursivo pode ser transformado em uma algoritmo iterativo. Tudo o que precisamos fazer é definir uma estrutura que armazene os dados necessários as chamadas recursivas e utilizar uma pilha para gerenciar estas estruturas. Desta forma não perdemos generalidade ao optar pela recursividade na apresentação.

O algoritmo básico de enumeração visita todos os conjuntos independentes de um grafo em ordem lexicográfica dos vértices. Assim sendo, considerando $V(G) = \{v_1, \ldots, v_n\}$, são gerados, recursivamente, todos os conjuntos independentes que contêm o vértice v_n como maior elemento antes daqueles que contêm v_{n-1} como maior elemento, e assim sucessivamente até gerar o conjunto independente $\{v_1\}$.

Definimos um *subproblema* como uma tripla ordenada (R, W, I) tal que $R \cap W = \emptyset$, $R \cup W = V(G)$ e I é um conjunto independente de $G[W]$. O conjunto R é chamado de *conjunto de candidatos*, o conjunto W de *conjunto de vértices visitados* e o conjunto independente I de *conjunto independente corrente*. Denotamos por $\alpha(R, W, I)$ valor $\alpha(G[R \cap \overline{N}(I)]) + |I|$.

A enumeração recursiva dos conjuntos independentes de um subproblema (R, W, I) é feita na função $ENUM()$ (Algoritmo 1). Para cada vértice r_i em $R = \{r_1, \ldots, r_{|R|}\}$, um novo subproblema é gerado tendo $(\{r_1, \ldots, r_{i-1}\} \cap \overline{N}(r_i))$ como conjunto de candidatos, $W \cup (\{r_1, \ldots, r_{i-1}\} \cap N(r_i))$ como conjunto de vértices visitados e $I + r_i$ como conjunto independente. Este procedimento é chamado de *visita* do vértice r_i. Observe que, devido a ordem estabelecida na linha 4, os vértices em R são visitados em ordem decrescente de índice. Veja também que a remoção dos elementos maiores que r_i de R na linha 5 é necessária para garantir que cada conjunto independente do grafo só seja produzido uma única vez durante a enumeração.

Na função $ENUM()$, as chamadas recursivas acontecem até que seja gerado um subproblema (R, W, I) cujo conjunto de candidatos seja vazio, o que indica que $W = V(G)$ e I é um conjunto independente de G de tamanho $\alpha(R, W, I)$.

Para determinar o valor do maior conjunto independente, empregamos uma variável global i_{max} que armazena o valor do maior conjunto independente conhecido em cada chamada recursiva de $ENUM()$. Para finalizar, observe que $\alpha(G)$ é obtido quando utilizamos a função $ENUM()$ com os dados $(V(G), \emptyset, \emptyset)$.

Os mapas de bits podem ser utilizados para representar todos os 3 conjuntos que definem um subproblema e utilizados também para representar os conjuntos de vizinhança dos vértices do grafo. Desta forma, todas as operações feitas com estes conjuntos são operações sobre mapas de bits e podem ser realizadas com operações lógicas sobre os nós destes mapas de bits, fazendo uso do paralelismo de bits. No Capítulo 2, detalhes sobre esse uso do paralelismo em bits são apresentados.

Na Figura 5 temos uma representação da árvore de visitação de subproblemas para uma execução do Algoritmo 1 para o grafo da Figura 4. Observe

Algoritmo 1: Enumeração recursiva de subproblema

```
1  ENUM(R, W, I)
   Entrada: Conjunto de candidatos R, conjunto de vértices visitados W e
            conjunto independente corrente I
   Saída: α(R, W, I)
2  se R = ∅ então retorna |I|
3  Seja R = {r_1, ..., r_|R|}
4  para i ← |R| até 1 faça
5      R ← R − r_i
6      W ← W + r_i
7      lim ← ENUM(R ∩ N̄(r_i), W ∪ N(r_i), I + r_i)
8      se lim > i_max então
9          i_max ← lim
10 retorna i_max
```

que o algoritmo $ENUM$ visita a árvore de subproblema em forma de uma busca em profundidade.

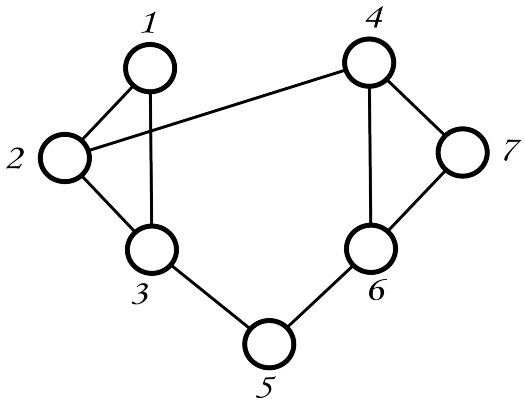


Figura 4: Grafo usado como exemplo.

O Algoritmo 1 é uma utilização do *método de backtracking* proposto em (AKKOYUNLY, 1973) e (BRON; KERBOSCH, 1973). A vantagem do método de *backtracking* é a eliminação de redundâncias na geração dos conjuntos independentes. Outro ponto importante é que estes algoritmos possuem necessidade de armazenamento polinomial durante a execução.

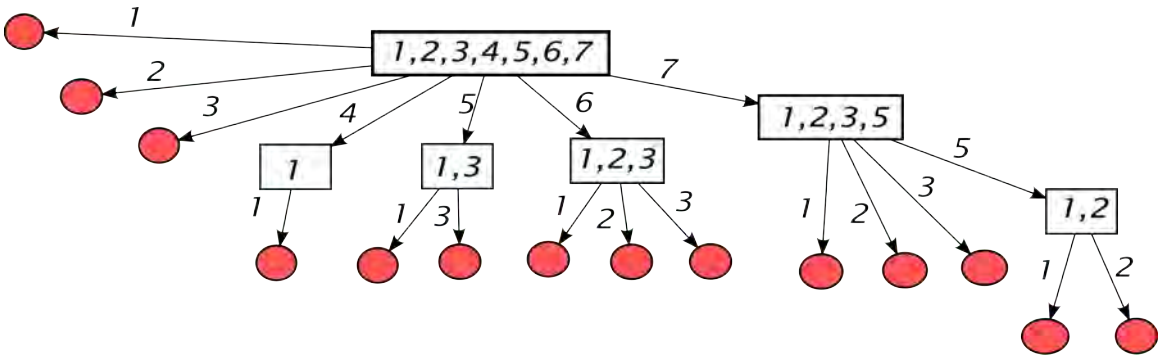


Figura 5: Árvore de subproblemas para o grafo da Figura 3.

1.6 Algoritmos Enumerativos com Poda para o Problema CIM

O Algoritmo 1 enumera todos os conjuntos independentes do grafo para determinar o valor de $\alpha(G)$. Entretanto, não é necessário enumerar conjuntos independentes *pequenos* se desejamos determinar a cardinalidade dos maiores conjuntos independentes. Tendo isso em mente, associamos a cada subproblema (R, W, I) um limite superior para $\alpha(R, W, I)$ que denotamos por $\ell(R, W, I)$, ou seja, $\ell(R, W, I) \geq \alpha(R, W, I)$.

Considere então uma chamada a $ENUM()$ com o subproblema (R, W, I). Se $\ell(R, W, I) \leq i_{max}$, então $\alpha(R, W, I) \leq i_{max}$ e os subproblemas que seriam gerados na visita de cada vértice de R não produziriam conjuntos independentes de tamanho maior que i_{max}. Neste caso, o subproblema (R, W, I) é prontamente eliminado da enumeração. Este procedimento é o que chamamos de *poda*.

Uma maneira simples de calcular um limite superior para um subproblema (R, W, I), arbitrário, é tomar $\ell(R, W, I) = |R| + |I|$, pois $\alpha(R, W, I) = \alpha(G[R \cap \overline{N}(I)]) + |I|$ e $|R| \geq \alpha(G[R \cap \overline{N}(I)])$. Na Figura 6 temos a indicação do que ocorre na árvore de subproblemas da Figura 5 quando utilizamos o critério de poda descrito acima. Os nós em cor mais clara não são gerados devido a podas.

Outra maneira de calcular um limite para $\alpha(G[R])$ é realizando uma coloração de $\overline{G}[R \cap \overline{N}(I)]$.

Observação 1. *Se $\overline{G}$ admite uma k-coloração, então $\alpha(G) \leq k$.*

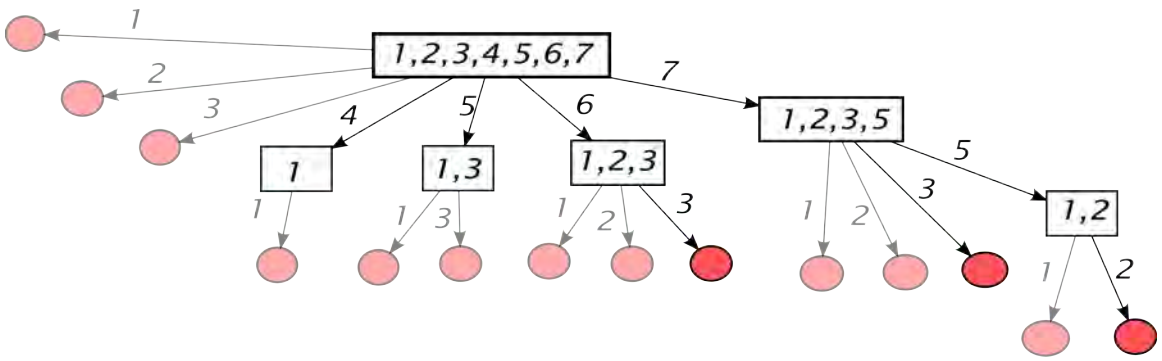


Figura 6: Árvore de subproblemas para o grafo da Figura 3 e poda através de $\ell(R, W, I) = |R| + |I|$.

Prova: Suponha por absurdo que $\alpha(G) > k$ e tome um conjunto independente máximo I e uma k-coloração de $\overline{G}$. Então, existem $u, v \in I$ tais que $c(v) = c(u)$ uma vez que $|I| > k$. Uma contradição ocorre visto que se $c(u) = c(v)$, então $uv \notin E(\overline{G})$ e logo $uv \in E(G)$. □

Observe que as classes de cor de uma k-coloração de $\overline{G}$ são cliques de G. Assim sendo, uma k-coloração de $\overline{G}$ é uma *cobertura por cliques* de G com k cliques.

A ideia da cobertura de cliques como limite superior para a enumeração de conjuntos independentes com poda foi testada com bons resultados em (TOMITA; KAMEDA, 2007). Posteriormente, os mesmos autores acoplaram um critério de recobertura, de maneira a tentar melhorar o limite superior obtido (TOMITA; AKUTSU; MATSUNAGA, 2009).

Estes algoritmos originalmente não utilizam mapas de bits para sua representação. Contudo, em (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011) é apresentada uma maneira de utilizar vetores de bits para representar os conjuntos de candidatos utilizados no algoritmo de enumeração com poda via cobertura de cliques.

Visto que o problema de coloração com um número mínimo de cores é um problema *NP-Difícil*, utiliza-se uma heurística gulosa para colorir $\overline{G}$ de complexidade $O(|V(G)|^2)$. Uma implementação judiciosa deste algoritmo com ma-

pas de bits pode reduzir o tempo de execução por um fator muito próximo de w quando comparada a uma implementação que não faça uso de paralelismo de bits (LEISERSON; PROKOP; RANDALL, 1998).

O algoritmo descrito em (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011) utiliza duas representações para o conjunto de candidatos R, a saber: um vetor de números inteiros, que é utilizado para armazenar a ordem na qual os vértices são visitados (ordenação de acordo com a cobertura por cliques) e um vetor de bits que é utilizado para realizar as operações sobre o conjunto de candidatos. É importante observar que o algoritmo descrito em (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011) precisa destas duas estruturas para atingir a eficiência descrita no artigo. O vetor de bits é necessário para acelerar as operações e o vetor de inteiros ordenados de acordo com o número das cliques ao qual cada vértice pertence é necessário para manter a verificação do critério de poda realizada em tempo constante para cada subproblema.

1.7 Método das Bonecas Russas

O Método das Bonecas Russas para resolver uma problema de maximização com restrições sobre um universo de n variáveis binárias, $x_1, \ldots, x_n$, consiste em resolver n subproblemas aninhados. O primeiro subproblema a ser resolvido é restrito a variável x_1. Uma vez obtida uma solução ótima do primeiro subproblema, o segundo subproblema definido com apenas as variáveis x_1 e x_2 é resolvido. A solução ótima do primeiro subproblema pode ser usada para acelerar a resolução do segundo subproblema. Este processo é repetido, com cada novo subproblema incorporando uma variável até que a resolução do último subproblema, que é o próprio problema original, se faça conhecendo-se uma solução ótima do subproblema definido com $x_1, \ldots, x_{n-1}$. Este método resumido pode ser encontrado em (VASKELAINEN, 2010). O artigo pioneiro neste método é (VERFAILLIE;LEMA;SCHIEX, 1996)

Esse tipo de abordagem foi utilizada em (ÖSTERGÅRD, 2002) para resolver o problema CIM da seguinte maneira: tomamos os vértices de $V(G) = \{v_1, \ldots, v_{|V(G)|}\}$ do maior para o menor. A fim de enquadrar o algoritmo na descri-

ção geral do Método das Bonecas Russas, imaginamos uma variável binária para cada vértice v embora essas não sejam explicitamente manipuladas no algoritmo. Ao examinarmos o vértice v_i, tentamos determinar, recursivamente, se existe um conjunto independente em $G[v_i, \ldots, v_{|V(G)|}]$ maior que o conjunto independente máximo, já conhecido, de $G[v_{i+1}, \ldots, v_{|V(G)|}]$. Observe que este conjunto, se existir, deve necessariamente conter v_i. Nos Algoritmos 2 e 3 temos uma representação da descrição feita acima. Uma observação relativa à linha 3 do Algoritmo 3: o valor ótimo para o subgrafo induzido por $\{v_1, \ldots, v_{|R|}\}$ pode ser usado como limite superior no teste para eliminação do subproblema no lugar de $|R|$.

Algoritmo 2: Descrição Geral Método das Bonecas Russas

Entrada: grafo G
Saída: $\alpha(G)$
1 Seja $V(G) = \{v_1, \ldots, v_{|V(G)|}\}$
2 $i_{max} \leftarrow 0$
3 **para** $i \leftarrow |V(G)|$ *até* 1 **faça**
4 $\quad k \leftarrow$ `ENUM_RUSSIANDOLLS(`$\{v_i, \ldots, v_{|V(G)|}\}$, $\emptyset$,i_{max}`)`
5 $\quad$ **se** $k = i_{max} + 1$ **então**
6 $\quad\quad i_{max} \leftarrow k$

Algoritmo 3: Ramificação e Subproblema no Método das Bonecas Russas

1 `ENUM_RUSSIANDOLLS(`R,I, lim`)`
Entrada: Conjunto de vértices R, Conjunto independente I e valor do maior conjunto independente conhecido lim
Saída: $\alpha(G)$
2 Seja $R = \{r_1, \ldots, r_{|R|}\}$
3 **se** $|R| < lim$ **então**
4 $\quad$ **retorna** $|I|$
5 **para** $i \leftarrow |R|$ *até* lim **faça**
6 $\quad I \leftarrow I + r_i$
7 $\quad$ **se** `ENUM_RUSSIANDOLLS(`$\{v_i, \ldots v_{|R|}\} \cap \overline{N}(r_i)$,$I$,$lim - 1$`)` $= lim$ **então**
8 $\quad\quad$ **retorna** *lim+1*
9 $\quad I \leftarrow I - r_i$
10 **retorna** lim

Na descrição proposta em (ÖSTERGÅRD, 2002), este algoritmo não utiliza mapas de bits para representar os conjuntos envolvidos. Este método será

melhor explorado no Capítulo 3 deste trabalho, no qual apresentamos um algoritmo, proposto em (CORRÊA;DONNE;LECUN;MAUTOR;MICHELON, 2013), que combina o Método das Bonecas Russas com poda por cobertura por cliques e paralelismo em bits.

1.8 Visão Geral do Texto

1.8.1 Escopo do Trabalho

Encontramos alguns trabalhos na literatura sobre vetores de bits e técnicas para realizar eficientemente operações sobre estes vetores (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011). No entanto, esses trabalhos têm enfoques em aplicações. Por essa razão, há pouca informação sobre a eficiência na prática do uso de tabelas de dispersão e estratificação. Uma referência que vale destacar é (DEMENTIEV;KETTNER;MEHNERT;SANDERS, 2004), na qual alguns experimentos são detalhadamente apresentados. No entanto, a estrutura proposta, que pode ser vista como uma variação da árvore de van Emde Boas, é voltada para a operação de busca de um elemento do conjunto. Não há estudos sobre operações lógicas nessas estruturas.

Neste trabalho, discutimos diferentes implementações para mapas de bits, além da tradicional implementação com vetor de bits, e examinamos como as técnicas aplicadas aos vetores de bits podem ser utilizadas nas demais implementações de mapas de bits para tornar estas implementações eficientes e econômicas em matéria de consumo de memória. Em particular, interessamo-nos pelo compromisso entre eficiência (em tempo) e consumo de memória, sobretudo em situações em que os conjuntos envolvidos são esparsos.

Analisamos o uso destas implementações em algoritmos enumerativos para o problema CIM. Para fazer isso, reimplementamos um algoritmo da literatura que faz uso de vetores de bits (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011) e acoplamos a interface de mapas de bits no lugar dos vetores e analisamos seu desempenho.

Ainda no tocante aos algoritmos para o problema CIM, aplicamos a

poda por cobertura de cliques ao método das bonecas russas e analisamos o impacto das diferentes implementações de mapas de bits sobre esta combinação.

Por fim, desejamos que as implementações citadas nos parágrafos anteriores sejam competitivas com as implementações com vetores de bits descrita na literatura, mas possam ser mais econômicas em termos de memória e que a implementação seja o mais flexível possível, permitindo a mudança da estrutura de dados sem prejudicar os algoritmos que as utilizam.

1.8.2 Organização do Texto

O texto está organizado da seguinte forma.

No Capítulo 2, apresentamos as estruturas de dados utilizadas e a técnica de paralelismo de bit e seu uso nas estruturas de dados e operações necessárias para os algoritmos de enumeração. Examinamos também a estrutura da implementação e alguns detalhes de codificação.

No Capítulo 3 examinamos com cuidado como integrar o método das bonecas russas com a poda por cobertura de cliques gerando o algoritmo que conheceremos por *RD_MaxStab*. Examinamos também como o paralelismo de bits é utilizado neste algoritmo.

Por fim, apresentamos os resultados computacionais obtidos com a implementação no Capítulo 4, onde discutimos os resultados em comparação com os demais resultados na literatura.

2 IMPLEMENTAÇÕES DE MAPAS DE BITS

Neste capítulo, apresentamos um visão geral do código da implementação de estruturas de dados para mapas de bits. Nos propomos a ter uma implementação não só eficiente como também flexível. Um dos meios que utilizamos para alcançar este objetivo é o uso de orientação a objetos para encapsular as estruturas de dados. Tratando-se de uma implementação sob o paradigma de orientação a objetos, a apresentação que segue destaca as classes e interações entre elas.

Ao final do capítulo, apresentamos também uma breve descrição das implementações de grafos e dos métodos de enumeração para o problema CIM.

2.1 Visão Geral da Implementação

São três as classes abstratas principais, relativas, respectivamente, a mapas de bits, grafos e enumeração com retrocesso. Nos dois primeiros casos, as classes derivadas definem estruturas de dados. Já no caso da classe abstrata de métodos de enumeração, as classes derivadas definem o tipo de enumeração.

Destacamos que a implementação, no que diz respeito aos mapas de bits, é flexível o suficiente para permitir a estratificação e a utilização de tabelas de dispersão para os nós. Respeitando o requisito de flexibilidade, procuramos desenvolver uma implementação o mais eficiente possível. Vale destacar ainda que a flexibilidade nos permite o reuso de código nos diferentes experimentos, cujos resultados estão descritos no Capítulo 4.

Nas seções posteriores, discutimos algumas implementações concretas das classes abstratas mencionadas. Nessa discussão, apontamos ainda as fontes de perda de eficiência com relação a implementações monolíticas das mesmas estruturas de dados.

A principal classe abstrata relacionada aos mapas de bits é denominada *BitMap*, na qual os métodos das operações sobre mapas de bits são definidos. Uma extensão desta classe abstrata é outra classe abstrata, *StratifiedBitMap*, que

identifica os mapas de bits estratificados.

As classes abstratas *BitMap* e *StratifiedBitMap* possuem 2 classes concretas cada uma, a saber *ArrayBitMap* e *PHashBitMap* para *BitMap*, e *StratifiedArrayBitMap* e *StratifiedPHashBitMap* para *StratifiedBitMap*. Na Figura 7 temos a representação resumida dessas classes e de suas relacões.

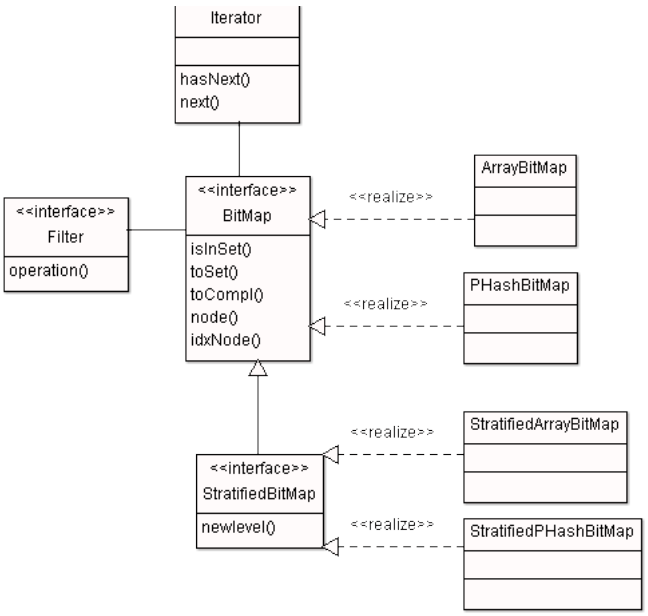


Figura 7: Diagrama de classes da porção do código relativa as estruturas de dados.

Utilizamos na exposição a mesma notação do Capítulo 1, mais especificamente, o universo de um mapa de bits é formado pelos elementos $\{0, \ldots, u-1\}$.

A linguagem escolhida para a implementação foi *C++*, por permitir tanto operações de baixo nível, em nível de bit, e abstrações de alto nível como classes e herança.

2.2 Classe Abstrata *BitMap*

Os métodos da classe abstrata *BitMap* são divididos em quatro grupos. A seguir, esses grupos são descritos através de alguns de seus principais métodos.

2.2.1 Caracterização da Estrutura de Dados

Este grupo concentra os métodos que devem ser implementados nas classes derivadas de *BitMap* a fim de implementar a estrutura de dados. Somente a classe abstrata *BitMap* tem acesso a estes métodos.

nodeIdx(x) retorna o índice do nó que armazena o bit associado ao elemento especificado.

nodeSize() retorna o número de bits em cada nó do mapa de bits.

idxInNode(x) retorna o índice, dentro do nó *nodeIdx(x)* correspondente ao bit associado ao elemento x.

node(i) retorna uma referência ao nó cujo índice é i.

Estes métodos permitem que quase todos os demais métodos e operações sobre conjuntos sejam feitas sem tomar conhecimento da estrutura de dados utilizada para representar o mapa de bits. A descrição de métodos dos demais grupos, feitas em seções posteriores deste capítulo, inclui exemplos de utilização dos métodos deste grupo.

2.2.2 Operações sobre Nós

Os métodos nesse grupo representam operações sobre nós, usualmente são operações para determinar o bit menos significativo de um nó. Nestes métodos utilizamos as funções *builtin*, ou uma delas, para acelerar a implementação destes métodos. Os dois principais métodos dessa classe são:

lsb(e) retorna a posição do bit menos significativo de valor 1 em e.

lsbAfter(e,i) retorna a posição do bit menos significativo de valor 1 em e, desconsiderando-se os bits de posições menores ou iguais a i.

Estas classes utilizam a função *builtin builtinLeastTYPE(e)*, que dado um nó do tipo de dados *TYPE*, retorna o valor do bit menos significativo neste nó.

Utilizando a orientação a objetos, podemos utilizar a função *builtinLeastTYPE(e)* sem nós preocuparmos com o tipo de dado associado ao nó. A função *lsb(e)* é apenas uma chamada direta a função *builtinLeastTYPE(e)*, e a função *lsbAfter(e,i)* é uma chamada a *builtinLeastTYPE(e)* com o cuidado de usar uma máscara de bits no nó antes de opera-lo para apagar os bits de posição menor ou igual a i. Esta máscara é construída para cada tipo de dados e para cada posição no inicio da execução do programa, o que não representa uma perda de eficiência por este número ser pequeno.

2.2.3 Operações sobre Elementos

Os métodos deste grupo são implementados na própria classe abstrata *BitMap* e envolvem operações que recebem um só elemento como parâmetro de entrada. Alguns destes métodos são os seguintes:

isInSet(x) verifica se o elemento x está no conjunto da seguinte forma:

retorna *node(*nodeIdx(x)*) & (1 « *idxInNode(x)*)

toSet(x) adiciona o elemento x ao conjunto.

$i \leftarrow$ *nodeIdx(x)*
node(i) = *node(i) | (1 « *idxIdNode(x)*)

toCompl(x) remove o elemento x do conjunto.

$i \leftarrow$ *nodeIdx(x)*
node(i) = *node(i) & (1 « *idxIdNode(x)*)

A notação **node(x)* indica o nó correspondente a referência *node(x)*.

2.2.4 Enumeração via Iteradores

A operação relevante que examinamos nesta subseção é a enumeração de todos os elementos de um conjunto. Acoplado à classe abstrata *BitMap*, existe uma classe abstrata *Iterator* que contém métodos para a enumeração do conjunto representado pelo mapa de bits. *Iterator* é um padrão de software que visa permitir a enumeração de um conjunto sem expor a sua representação interna. A classe *BitMap* possui um método, *newIterator* que permite obter um iterador padrão para a estrutura sem saber como esta é realmente implementada . Neste contexto, o padrão *Iterator* é descrito, usualmente, como uma interface que apresenta dois métodos, a saber:

hasNext(): verifica se ainda existe algum elemento não visitado no conjunto.

next(): retorna o próximo elemento a ser visitado do conjunto, se este existir. Caso não exista um elemento não visitado no conjunto, é retornado um valor não pertencente ao conjunto. Em nosso caso um inteiro $p \geq u$.

Veja que os métodos que caraterizam a estrutura de dados podem ser usados para descrever os métodos do *Iterator* sem menção explícita à estrutura de dados realmente utilizada para representar o mapa de bits. Estas operações são apresentadas no Algoritmo 4. Algumas implementações de *BitMap* podem possuir maneiras mais rápidas de realizar esta enumeração, como é o caso de *StratifiedBitMap*. Estas devem, então, sobrescrever o *Iterator* padrão de *BitMap*.

O iterador padrão para *BitMap* armazena o nó corrente na enumeração, *node*, a posição deste nó no mapa de bits, *inode*, o índice, em *node*, do elemento corrente na enumeração, $cursor$ e o índice do primeiro elemento e *node* na enumeração em curso, *dspl*. Todas estas variáveis são inicializadas na construção do iterador por uma função chamada de *init()*.

A segunda observação diz respeito ao tratamento da possibilidade de desalinhamento entre os conjuntos envolvidos na operação. Um exemplo de possível desalinhamento é mencionado na Subseção 2.5. Um conjunto é *alinhado* quando $idxInNode(0) = 0$. Caso contrário, o conjunto é *desalinhado*. O método *logicO-*

Algoritmo 4: *Iterator* padrão para *BitMap*

```
   Entrada: mapa de bits B_A
 1   inode ← 0
 2   dspl ← 0
 3   node ← 0
 4 advance()
 5      cursor ← lsbAfter(*node, cursor)
 6      nextdspl ← nodeSize() ≪ 3
 7       enquanto cursor < 0 e nextdspl < u faça
 8          dspl ← nextdspl
 9          inode ← inode +1
10          node ← node(inode)
11          cursor ← lsb(*node)
12          nextdspl ← nextdspl+ nodeSize() ≪ 3
13      se cursor < 0 então
14          dspl ← nextdspl
15          cursor ← 0
16 init()
17   cursor ← idxInNode(0)
18   inode ← node(0)
19   node ← node(inode)
20   dspl ← cursor
21   se *node & (1LL ≪ cursor ) = 0 então
22    advance()
23 hasNext()
24      retorna dspl + cursor ≤ u − 1
25 next()
26      dspl ← dspl + cursor
27      x ← dspl
28      advance()
29      retorna x
```

peration deve lidar com a possibilidade de desalinhamento de quaisquer dos 3 conjuntos envolvidos.

2.2.5 Operações entre Conjuntos

Operações que se beneficiam do paralelismo de bits aparecem neste grupo de métodos que agem sobre pares de conjuntos. Algumas destas operações são realizadas pelos seguintes métodos:

intersectionOf(x, y, z) realiza a interseção entre os mapas de bits y e z e armazena o resultado no conjunto x.

retainAll(x) realiza a interseção deste mapas de bits com o mapa de bits passado como parâmetro.

unionOf(x, y, z) Realiza a união entre os mapas de bits y e z e armazena o resultado no mapa de bits x.

diffOf(x, y, z) Realiza a diferença entre os mapas de bits y e z e armazena o resultado no mapa de bits x.

symDiffOf(x, y, z) Realiza a diferença simétrica entre os mapas de bits y e z e armazena o resultado no mapa de bits x.

Há duas observações revelante com respeito à implementação destes métodos. A primeira observação é que todos são indireções para o seguinte método privado da classe abstrata *BitMap*:

$$LogicOperation(rs,\ s_1,\ s_2,\ f)$$

O parâmetro f é um instância da interface *Filter*, indicada na Figura 9. O único método desta interface é:

$$Filter(resNode,\ node1,\ node1)$$

o qual efetua uma operação lógica entre os nós cujas referências são $node1$ e $node2$ atribuindo o resultado ao nó cujo referência é $resNode$. A classe abstrata *BitMap* mantém uma instância desta interface para cada operação lógica efetuada no grupo de operações entre conjuntos. Por exemplo, o método *filter* da instância de operações de E lógico é implementado da seguinte forma:

$$*res \leftarrow *node1 \ \& \ *node2$$

No Algoritmo 5, apresentamos um esqueleto dos métodos de união, interseção, negação e subtração de conjuntos para mapas de bits sem desalinhamento. Omitimos o caso com alinhamento por ser mais complexo.

Algoritmo 5: Esqueleto para realização de operações de conjuntos com *BitMap*

Entrada: *BitMap* A, *BitMap* B, *BitMap* C e *Filter* OP

```
1 para i ← 0 até nodeIdx(u-1) faça
2     OP.filter(C.node(i), A.node(i), B.node(i))
```

2.3 Classe Abstrata *StratifiedBitMap*

Um *StratifiedBitMap* é uma estrutura recursiva formada por um *BitMap* B e um *StratifiedBiTMap* S que funciona como o sumário do *BitMap* B. Assim sendo, entanto o *BitMap* B tem o conjunto $\{0, \ldots, u-1\}$ como universo, o *StratifiedBitMap* S tem como universo o conjunto $\{0, \ldots, \lfloor \frac{u}{w} \rfloor\}$ e possui como elementos os índices dos nós de B que não são nulos.

Um método adicional da classe abstrata é a criação de um novo nível no mapa de bits estratificado, *newLevel(FactoryBitMap*)*, que cria um novo nível de sumário na estrutura.

A seguir, as modificações que os métodos devem sofrer para lidar com os diferentes níveis da estrutura estratificada são apresentadas segundo os grupos estabelecidos para *BitMap*.

2.3.1 Operações sobre elementos

As operações de inclusão ou remoção de elementos apresentam algumas diferenças com relação à implementação em *BitMap*, para manter a semântica correta da estrutura de sumário.

toSet(x) deve verificar se o nó no qual o elemento x foi incluído era vazio e se tornou não vazio para, recursivamente, atualizar o sumário.

$i \leftarrow$ *nodeIdx(x)*
node(i) = *node(i) | (1 $\ll$ *idxIdNode(x)*)
$S.toSet(i)$

toCompl(x) precisa verificar se o nó se tornou vazio pela remoção do elemento x e atualizar o sumário.

$i \leftarrow$ *nodeIdx(x)*
node(i) = *node(i) & (1 $\ll$ *idxIdNode(x)*)
Se(*node(i) = 0) $S.toCompl(i)$

isInSet(x) é feita diretamente sobre o *BitMap* B.

2.3.2 Enumeração via Iteradores

O iterador de *StratifiedBitMap* funciona de forma semelhante ao iterador de *BitMap*, com a diferença que utiliza o sumário e o iterador deste para acelerar a enumeração. O iterador de *StratifiedBitMap* armazena uma referência para os iterador do sumário, itS.

A operação de inicialização das variáveis e busca do primeiro elemento do conjunto é bem simples e consiste apenas na chamada da função $init()$ do iterador do sumário e da atualização das demais variáveis.

O método $next()$ realiza a busca pelo próximo elemento do conjunto. Este método funciona da seguinte maneira: seja x o último elemento enumerado

do conjunto. Inicialmente procuramos no nó que contém x pelo próximo elemento. Se este não for encontrado usamos o iterador do sumário para determinar, recursivamente, o próximo nó não nulo e retornamos o primeiro elemento deste nó. No Algoritmo 6 apresentamos o código que corresponde a esta ideia.

Algoritmo 6: *Iterator* Padrão para *StratifiedBitMap*

```
   Entrada: mapa de bits estratificado B_A^S
 1   inode ← 0
 2   dspl ← 0
 3   node ← 0
 4   itS ← iterador para sumário de B_A^S
 5 advance()
 6       cursor ← lsbAfter(*node, cursor)
 7       inode ← itSnext()
 8       dspl ← inode * nodeSize()
 9       node ← node(inode)
10       cursor ← lsb(*node)
11       se cursor < 0 então
12           dspl ← nextdspl
13           cursor ← 0
14 init()
15   cursor ← idxInNode(0)
16   inode ← node(0)
17   node ← node(inode)
18   dspl ← cursor
19   itS.init()
20   se *node & (1LL ≪ cursor) = 0 então
21    advance()
22 hasNext()
23       retorna dspl + cursor ≤ u − 1
24 next()
25       dspl ← dspl + cursor
26       x ← dspl
27       advance()
28       retorna x
```

As operações sobre conjuntos nesta estrutura, assim como sobre a interface *BitMap* são feitas através de objetos da interface *Filter*, o percurso entretanto

é feito com o uso do sumário e dos objetos do tipo *Iterator* podemos evitar operar palavras nulas, proporcionando uma ganho de tempo.

2.4 Classes Derivadas de *BitMap* e *StratifiedBitMap*

2.4.1 *ArrayBitMap* e *StratifiedArrayBitMap*

A primeira implementação concreta de *BitMap* emprega um vetor de nós T como estrutura de dados, dando origem a classe concreta *ArrayBitMap*. De forma semalhante *StratifiedArrayBitMap* que implementa *StratifiedBitMap*, emprega um vetor de nós como estrutura de dados ao usar uma instâcnia de *ArrayBitMap* em B.

A caracterização do vetor de nós através dos métodos $node()$, $nodeIdx()$ e $idxInNode()$ é a seguinte:

Seja T um vetor de nós,

- $node(i) \equiv retorna\ \&T[i]$,
- $nodeIdx(i) \equiv i \ll \log w$ e
- $idxInNode(i) \equiv i\ \&\ mask$. $mask$ é um nó cujo $\log w$ bits menos significativos são 1 e todos os demais são zero.

A flexibiliade proporcionada pela adoção do paradigma de orientação a objetos traz um custo adiconal, quando comaprada à implemntação monolítica, devido a indireções nos acessos ao vetor de nós. Uma forma de captar a diferença causada por esses fatores é o experimento no qual mede-se o tempo de enumeração dos 100 primeiros elementos de um conjunto gerado aleatoriamente para um universo de tamanho 5000, com diferentes densidades. Para medir o tempo do *ArrayBitMap*, realiza-se a enumeração com o código abaixo, considerando que *bitMap* seja uma instância de *BitMap* com o tipo *unsigned long long* para cada nó.

```
it = bitMap->newIterator();
for (i = 0; i < 100; i++)
```

```
  it->next();
delete it;
```

Por outro lado, para o uso de um vetor de bits, utiliza-se o código

```
typedef unsigned long long BITARRAY;
BITARRAY bitvector[NODESIUNIVERSE];

w = 0;
vec = bitvector[w];
b = -1;
for (i = 0; i < 100; i++)
  do {
    b = __builtin_ffsll(vec & __mask[b+1])-1;
    if (b < 0)
      vec = bitvector[++w];
  } while (b < 0);
```

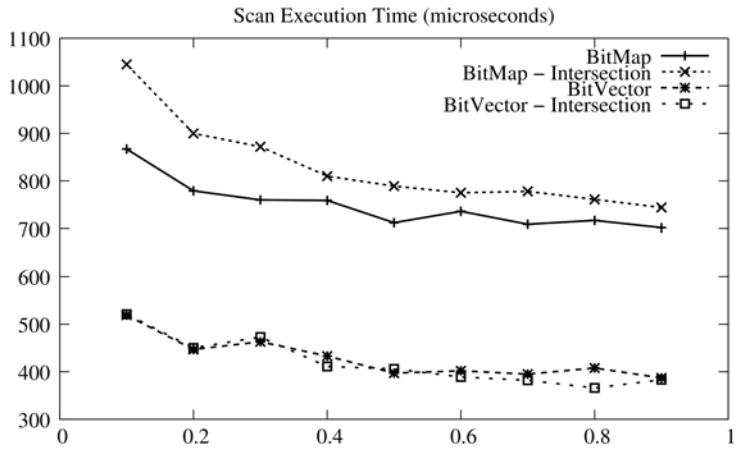


Figura 8: Tempo de enumeração dos 100 primeiros elementos de um conjunto formado aleatoriamente em um universo de tamanho 5000. Para cada caso, há dois cenários para a enumeração: no primeiro, o conjunto é enumerado diretamente, enquanto no segundo a enumeração é feita após a interseção do conjunto com outro igual a ele. Os tempos apresentados correspondem a 1000 execuções de cada enumeração, sendo que para cada uma um novo conjunto é gerado aleatoriamente.

As medições de tempo de execução nesse experimento são apresentadas no gráfico da Figura 8. A partir desses valores, observa-se que o custo incorrido por *ArrayBitMap* é aproximadamente 2 vezes o da implementação monolítica.

2.4.2 *PHashBitMap* e *StratifiedPHashMap*

Outra implementação para a interface *BitMap* utiliza também a ideia de um vetor de bits característico com o uso adicional de uma *tabela de dispersão* para não armazenar nós vazios na representação dos elementos que não pertencem ao conjunto.

Utilizamos uma tabela de dispersão minimal e perfeita como descrita em (BELAZZOUGUI; BOTELHO; DITZFELBINGER, 2009). Obtemos assim a classe concreta *PHashBitMap*. No código, utilizamos a biblioteca *cmph (C Minimal Perfect Hashing Library)* que inclui funções de construção (*cmph_new*) e para busca na tabela (*cmph_search*).

Em particular, a função de construção da tabela dependente do conjunto inicial sobre o qual a tabela de dispersão deve ser criada. Como consequência, esta estrutura é *estática* no sentido que não podemos realizar inclusões/remoções uma vez a tabela criada, impossibilitando a implementação dos métodos *toSet* e o *toCompl*. Esta classe é implementada como uma extensão de *ArrayBitMap* cujo universo é $m + 1$. Os nós de T de índices entre 0 e $m - 1$ formam a tabela de dispersão. O nó de T de índice m é preenchido com 0 e é usado para preencher os nós nulos não representados na tabela.

O método *nodeIdx* é reescrito na forma de uma função de dispersão e sofre um custo adicional devido ao uso de dispersão perfeita. Vale ressaltar que este custo é considerável, uma vez que é adicionado a cada acesso a um nó da tabela de dispersão.

Na implementação de *nodeIdx* existe a necessidade de uma lista de inteiros *nodeId* para identificar os nós não vazios em T. Isso ocorre porque a função de dispersão pode retornar, para elementos que não estão no conjunto, o valor de um nó válido. Logo, precisamos da informação que o nó ao qual pertence o elemento é na verdade vazio. O iterador para esta estrutura também apresenta uma

pequena modificação. Na linha 4 da função *advance()* do Algoritmo 4 o incremento inode ← inode+1 deve ser substituído por inode ← *nodeIdx*(dspl).

$i \leftarrow$ ArrayBitMap::*nodeIdx*(x)
$j \leftarrow$ cmph_search(i)
Se(nodeId[j] = i) *retorna* j
retorna m

A classe *StratifiedPHashBitMap* não necessita da tabela de inteiros $nodeId$, uma vez que o sumário realiza a função da tabela indexando os nós não vazios.

$i \leftarrow$ ArrayBitMap::*nodeIdx*(x)
$j \leftarrow$ cmph_search(i)
Se(Sumario.*isInSet*(j)) *retorna* j
retorna m

2.5 Cálculo do Consumo de Memória

Na próximo capítulo realizamos experimentos sobre as implementações concretas da interface *BitMap* e um dos aspectos mais importantes que analisamos é o consumo de memória.

De modo a deixar mais claro os resultados apresentados no próximo capítulo, explicamos agora como a medição de consumo de memória é realizada para cada estrutura. Procuramos medir apenas o consumo de memória nas classes concretas apenas de estruturas que variam de acordo com o tamanho e/ou esparsidade do conjunto; estruturas de tamanho fixo, como variáveis inteiras ou referências a outros objetos são desconsiderados.

Para a classe *ArrayBitMap* o consumo de memória é o número de bits usados no vetor de bits. Todas as demais informações armazenadas pela classe são desconsideradas por serem estáticas.

Paras as classes que implementam *StratifiedBitMap*, o consumo de memória é a soma do consumo do *BitMap* de base mais o consumo do sumário. Como já dito antes o consumo de memória com estruturas auxiliares é desconsiderado em cada nível. Aqui podemos justificar isso afirmando que este consumo é desprezível uma vez que esta estrutura, como dito no Capítulo 1 não possui uma altura muito grande, usualmente 2 ou 3.

Para *PHashBitMap* o consumo de memória é representado pelo número de bits usados na tabela de dispersão e na representação da função de dispersão, assim como o vetor de inteiros para o uso da função $isInSet()$. Lembre-se que como dito antes no caso do uso da estratificação o vetor de bits é desnecessário.

2.6 Classe Abstrata *Graph*

Não embutimos na classe abstrata *Graph* uma estrutura de dados para representar as arestas. Ao invés disto, usamos os objetos das classes concretas que implementam a classe abstrata *BitMap* para representar o conjunto de arestas do grafo. Neste contexto, temos duas classes concretas que implementam *Graph*, dependendo do número de mapas de bits utilizados para representar o conjunto de aresta do grafo.

A classe abstrata *Graph* apresenta dois métodos principais:

hasEdge(i,j) verifica se existe uma aresta entre i e j.

neig(i) retorna a vizinhança de um vértice como um mapa de bits.

addEdge(i,j) adiciona a aresta ij ao grafo.

removeEdge(i,j) remove a aresta ij do grafo. As demais operações são a inclusão e remoção.

2.6.1 A Classe Abstrata *BitMapsGraph*

Se utilizamos um mapa de bits para representar cada vizinhança, sob a forma de um conjunto onde apenas pertencem ao conjunto os vizinhos de cada

vértice, temos a classe *BitMapsGraph*.

Os métodos usuais sobre grafos são implementados de maneira bem simples nesta classe, em especial, o método $hasEdge(i, j)$ apenas verifica a pertinência do elemento j ao mapa de bits $neig(i)$ que é o próprio mapa de bits usado para representar a vizinhança do vértice i.

2.6.2 A Classe Abstrata *BitMapGraph*

Entretanto, se utilizamos apenas um mapa de bits para representar o conjunto de aretas, ou seja, o mapa de bits indexa as arestas e armazenas as arestas que pertencem ao grafo, temos a classe *BitMapGraph*.

Na classe *BitMapGraph*, o *BitMap* representa todas as possíveis arestas (que são listadas de acordo com o primeiro vértice de cada par-ordenado, assim sendo $V(G) = \{v_1, \ldots, v_n\}$ o conjunto de vértices do grafo, inicialmente listamos todos as os pares ordenados da forma v_1y para $y \in V(G)$ e depois todos da forma v_2y e assim por diante) e assim a operação $hasEdge(i, j)$ é implementada com a própria operação $isInSet()$ do mapa de bits. Entretanto, a operação $neig(i)$ apresenta o problema de ter q lidar com mapas de bits deslocados, a parte do mapa de bits de todas as arestas que representa as arestas de um vértice pode não começar no início de um nó (entretanto as arestas uv referentes ao mesmo u estão agrupadas de maneira contínua da maneira como numeramos as arestas do grafo). Desta maneira é necessário algumas operações para alinhar as vizinhanças de maneira a permitir que as operações lógicas possam ser realizadas nó a nó.

2.7 Algoritmos Enumerativos com Poda

A árvore de busca é representada pela classe *BackTracking*, esta classe armazena a lista de subproblemas e implementa todo o processo de manipulação desta lista e de enumeração, a exceção dos critérios de poda e de ramificação de subproblemas. Estes critérios são implementados por um objeto de uma das implementações concretas da classe abstrata *Modeler*, que implementa também métodos de alocação de memória para as estruturas que representam os subproblemas. As

estruturas de subproblema variam de acordo com o algoritmo de enumeração utilizado, isto para economizar memória e embutir nas estruturas algumas informações adicionais úteis aos algoritmos implementados.

Dito isto, o fluxo do programa é realmente bem simples como descrito a seguir:

- Criamos um objeto da classe *Graph* para representar o grafo de entrada. Utilizando a estrutura de dados que mais achamos adequada.
- Criamos um objeto da classe *BackTracking* para realizar a enumeração.
- Instanciamos um algoritmo de enumeração, utilizando uma classe concreta, filha da classe *Modeler*.
- O objeto da classe *Modeler* cria o subproblema raiz, que é incluído na lista de subproblemas do objeto da classe *BackTracking*.
- Enquanto a lista de subproblemas não for vazia, o objeto da classe *BackTracking* retira um subproblema da lista e utilizando o objeto da classe *Modeler*, tenta podar e ramificar o subproblema. Se a ramificação é possível, os filhos são incluídos na lista de subproblemas.

Um programador mais experiente pode ter observado que as diferentes implementações da interface *Modeler* utilizam, também, diferentes tipos de estruturas de subproblema. Desta maneira, a classe *BackTracking* devia possuir uma implementação para cada tipo, ou existir uma interface que unificasse todas as diferentes estruturas de subproblema.

Resolvemos este problema de uma maneira diferente. Em C++ podemos armazenar um ponteiro, referência para uma posição de memória, de um tipo abstrato de dados (*void*). A classe *Backtracking* trata os subproblemas como ponteiros *void* e apenas as implementações concretas de *Modeler* sabem o tipo de estrutura de subproblema utilizado.

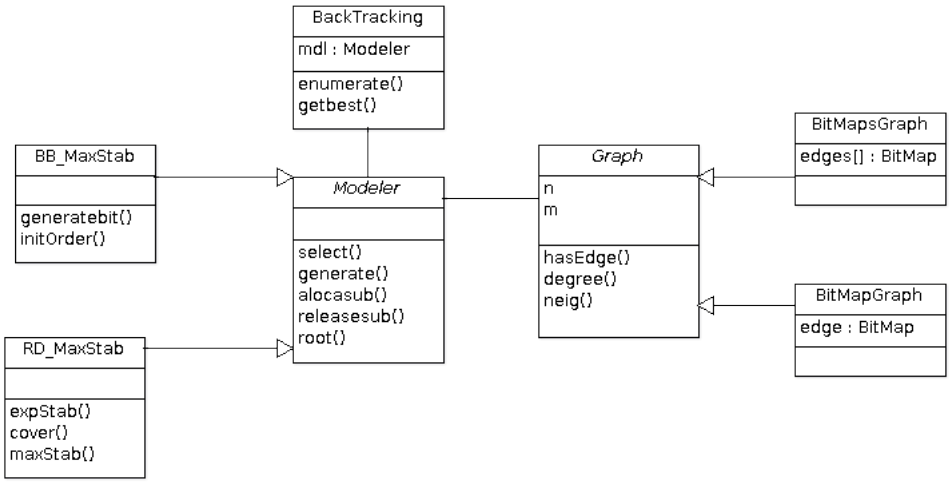


Figura 9: Diagrama de classes da porção do código relativa a enumeração.

3 CONJUNTO INDEPENDENTE MÁXIMO VIA MÉTODO DAS BONECAS RUSSAS

Neste capítulo, apresentamos um novo algoritmo proposto em (CORRÊA;DONNE;LECUN;MAUTOR;MICHELON, 2013) combinando o Método das Bonecas Russas (ÖSTERGÅRD, 2002) com poda por cobertura por cliques (TOMITA; KAMEDA, 2007) e paralelismo de bits (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011). Desta forma, o algoritmo apresentado neste capítulo pode ser visto como uma combinação de técnicas já aplicadas com sucesso ao problema CIM. Este é o algoritmo *RD_MaxStab*. Apresentamos também como a utilização de mapas de bits ocorre no algoritmo.

3.1 Ordenação dos Vértices

Seguindo o princípio do Método das Bonecas Russas, a primeira etapa do algoritmo consiste em estabelecer uma renumeração dos vértices do grafo para, com ela, definir a ordem em que os vértices são examinados durante a execução de heurísticas gulosas. O critério de ordenação adotado segue a proposta apresentada em (TOMITA; KAMEDA, 2007) (adaptando-a para o problema *CIM*), em que cada vértice i, com $i \in [n]$, possui o maior grau em $G[i]$, o subgrafo induzido por $[i]$ em G. No caso em que um vértice $j \in [i-1]$ possua, em $G[i]$, o mesmo número de vizinhos que i, então a escolha de i se faz por um segundo parâmetro dado pale soma dos graus dos vizinhos. Mais especificamente, se $deg_{G[i]}(i) = deg_{G[i](j)}$, então o vértice i é tal que $deg_{G[i]}(N_{G[i]}(i)) \geq deg_{G[i]}(N(G[i])(j))$.

O custo de tempo de ordenação dos vértices ocorre apenas uma vez, pois a ordenação determinada inicialmente não é mais alterada. De fato, essa ordenação é usada para enumerar os vértices na sua representação como mapas de bits na forma da classe *BitMapGraph* implementada com alguma classe derivada de *BitMap* como estrutura de dados. Vale ressaltar que nessa implementação, o método *neig* permite acesso à vizinhança de um vértice sob a forma de mapa de bits. Esse fato é explorado em vários pontos do algoritmo, conforme destacado nas

subseções seguintes.

3.2 Cobertura por Cliques e Paralelismo em Bits

De forma similar ao que é estabelecido no Algoritmo 3, um subproblema é caracterizado por um subgrafo de G. No caso do Algoritmo 3, o subgrafo associado a cada vértice i é $G[\{i, i+1, \ldots, n\}]$. Portanto, a numeração dos vértices determina a sequência em que os subproblemas são examinados (a notar que a numeração dos vértices no Algoritmo *new* de (ÖSTERGÅRD, 2002), do qual o Algoritmo 3 é um resumo, segue um critério diferente daquele apresentado na Subseção 3.1). Em *RD_MaxStab*, porém, tomamos os subgrafos em uma ordem correspondente a uma cobertura por cliques de G. Outra particularidade de *RD_MaxStab* é que alguns subproblemas são eliminados sem a necessidade de serem examinados. Embora os detalhes desse processo sejam descritos nas seções seguintes, podemos adiantar que essa eliminação de um subproblema G_k depende do limite superior $ub(G_k)$ para $\alpha(G_k)$ decorrente da cobertura por cliques de G_k. Passamos, então, a descrever a forma que adotamos para a determinação de uma cobertura por cliques de um subproblema G_k arbitrário com o uso de paralelismo em bits.

Algoritmo 7: Guloso de cobertura por cliques

Entrada: grafo G, conjunto R, conjunto W, número *lim* máximo de cliques
Saída: número de vértices cobertos pelas cliques

```
1  coberturaVert(G,R,W,lim) cor ← 1
2  ver ← 0
3  enquanto S ≠ ∅ e cor ≤ lim faça
4      Γ ← copyFrom(R)
5      enquanto Γ ≠ ∅ faça
6          v ← menor vértice de Γ
7          R− > toCompl(v)
8          W− > toSet(v)
9          ver ← ver + 1
10         Γ− > retainAll(neig(v))
11 retorna ver
```

A heurística *cover* apresentada no Algoritmo 7 é uma versão para mapas

de bits da heurística gulosa por cores descrita no Apêndice A (Algoritmo 11). Além do grafo G representado sob a forma de uma instância de *BitMapGraph*, essa heurística tem como entrada três outros parâmetros. Os dois primeiros são mapas de bits, contendo o primeiro deles os vértices de G que definem subgrafo envolvido, enquanto o segundo deve receber os vértices efetivamente incluídos em alguma clique da cobertura. O terceiro parâmetro é o número máximo lim de cliques admitidas na cobertura. O algoritmo usa ainda um mapa de bits temporário Γ. Os dois pontos que merecem atenção são o potencial uso de paralelismo de bits (o seu uso efetivo depende da estrutura de dados usada para implementar o grafo G e os conjuntos envolvidos) nas chamadas a *copyFrom* (Linha 7) e *retainAll* (Linha 7) e a interrupção das iterações assim que o número de cliques atinge o valor máximo lim.

Outras duas heurísticas utilizadas no algoritmo proposto são descritas a seguir, a primeira delas é uma heurística para determinar um conjunto independente maximal (Algoritmo 8) que contém um conjunto independente já conhecido e a segunda é utilizada para determinar um clique maximal (Algoritmo 9). Em ambas as heurísticas todos os conjuntos envolvidos são representados como mapas de bits. O principio de ambas as heurísticas é o mesmo a única diferença está no uso da operação *retainAll* ou *removeAll* para atualizar o conjunto de vértices candidatos a entrarem no conjunto.

3.3 Primeira Versão: Cobertura por Cliques Estática

Para fins de maior clareza de apresentação, descrevemos a seguir uma versão simplificada do Algoritmo *RD_MaxStab*. A simplificação existente nessa versão está no fato de o processo de enumeração ser baseado em uma cobertura por cliques de G estabelecida no princípio da enumeração ser baseado em uma cobertura por cliques de G estabelecida no começo da enumeração e mantida até o final. O papel dessa cobertura é definir a ordem dos vértices usada para a definição da sequência de percurso dos subproblemas. Para determinar a cobertura, pode-se usar uma adaptação do Algoritmo 7 que use um vetor como conjunto W (e não um mapa de bits) e que disponha os vértices em W na ordem em que eles são inseridos

Algoritmo 8: heurística para conjunto independente maximal

Entrada: grafo G, conjunto independente I, conjunto de vértices R
Saída: conjunto independente maximal I^* tal que $I \subseteq I^*$

```
1  conjuntoInd(G,R,W)
2  I ← ∅
3  Γ ← W
4  for v ∈ I do
5      Γ− > removeAll(neig(v))
6  enquanto Γ ≠ ∅ faça
7      v ← menor vértice de Γ
8      Γ− > toCompl(v)
9      I*− > toSet(v)
10     ver ← ver + 1
11     Γ− > removeAll(neig(v))
12 retorna ver
```

Algoritmo 9: heurística para clique maximal

Entrada: grafo G, conjunto de vértices R
Saída: clique maximal $C \subseteq R$

```
1  cliqueMaximal(G,R)
2  C ← ∅
3  Γ ← R
4  enquanto Γ ≠ ∅ faça
5      v ← menor vértice de Γ
6      Γ− > toCompl(v)
7      C− > toSet(v)
8      Γ− > retainAll(neig(v))
9  retorna C
```

nas cliques. Dessa forma, os primeiros vértices em W seriam aqueles da primeira clique formada, seguidos daqueles da segunda clique, e assim por diante. Mais especificamente, seja $c : [n] \rightarrow \{1, \ldots, k\}$ o mapeamento que descreve a cobertura por cliques calculada com a heurística gulosa modificada. Então, os vértices são dispostos em W e forma que $c(u) < c(v)$ implica que o índice de u em W é menor que o índice de v. Além dessa alteração no papel desempenhado por W, a heurística gulosa modificada deve também calcular um novo vetor, digamos UB, contendo $UB[i] = c(W[i])$, para todo $i \in [n]$.

A aplicação do Método das Bonecas Russas como uma modificação do Algoritmo 2 é feita sobre dois vetores W e UB construídos durante a execução inicial da heurística gulosa de cobertura por cliques descrita acima. A sequência de subproblemas é dada por $\langle G_1, G_2, \ldots, G_n \rangle$ tal que, para todo $i \in [i]$, o subgrafo G_i é o subgrafo de G induzido por $\{W[1], \ldots, W[i]\}$. O critério usado para eliminação de subproblemas é baseado em UB e no valor MAX do maior conjunto independente de G conhecido. O valor inicial de MAX é obtido com a heurística gulosa descrita no Algoritmo 8. O fato importante a ser destacado é que esses dois parâmetros, UB e MAX, permitem a aplicação da Observação 1 para eliminar da enumeração todos os subproblemas G_i tais que $UB[i] \leq MAX$.

O algoritmo é mostrado no Algoritmo 10 no qual omitimos a chamada para a verificação de $\alpha(G_i[\bar{N}(W_i)])$ por simplicidade. Essa verificação deve decidir se G_i possui um conjunto independente de tamanho $MAX + 1$, sabendo que $\alpha(G_{i-1}) = MAX$. Note que se a resposta a essa verificação é positiva, então todo conjunto independente máximo de G_i inclui o vértice i. Observe também que a sequência em que os subproblemas são examinados na Algoritmo 10 é a sequência inversa daquela seguida pelos Algoritmos MCR ((TOMITA; KAMEDA, 2007)) e *BB_MaxClique* ((SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011))

O Algoritmo *RD_MaxStab* difere do Algoritmo 10 em dois pontos principais. O primeiro é a definição dos subproblemas, e a sequência em que são examinados. Em *RD_MaxStab*, a cobertura por cliques não é inteiramente determinada no início da enumeração, mas sim construída na medida em que a enumeração avança. A sequência assim obtida pode diferir da sequência do Algoritmo 10. O segundo ponto de diferença é o critério de eliminação de subproblemas. No Al-

Algoritmo 10: Método das Bonecas Russas Simplificado

```
1 Determine W e UB com heurísticas gulosas de cobertura por cliques
2 Determine o valor inicial de MAX com Algoritmo 8
3 i ← menor índice tal que i ≤ n + 1 ou UB[i] > MAX
4 enquanto i ≤ n faça
5     se α(G_i[N̄(W[i])]) = MAX então
6         MAX ← MAX + 1
7         Incremente i até o menor índice tal que i ≤ n + 1 ou UB[i] > MAX
8     senão
9         i ← i + 1
```

goritmo 10, essa eliminação ocorre nas linhas 3 e 7. Em *RD_MaxStab* um limite superior baseado em uma cobertura por cliques é potencialmente, o número de subproblemas eliminados.

3.4 Sequência de Subproblemas

Em *RD_MaxStab*, definimos n de subgrafos $G_1, \ldots G_n$ para cujos subproblemas são determinados limites superiores. Para definir os subgrafos, e a exemplo do que é feito no Algoritmo 10 determina-se inicialmente uma solução viável, estabelecendo assim o valor MAX da melhor solução já encontrada. Cada um dos subproblemas é definido a partir de um subgrafo $G_k = (W_k, E_k)$ maximal quanto à propriedade $ub(G_k) \leq MAX$, onde $ub(G_k)$ é um limite superior (baseado em uma cobertura de cliques, como se verá adiante) para G_k. Para $k \geq 2$, além de maximal, a condição $W_{k-1} \subseteq W_k$ deve ser satisfeita.

Os subproblemas são gerados e examinados consecutivamente. Realizamos uma cobertura por cliques $C_1, \ldots, C_{MAX}$ dos vértices de G. Retiramos então todos os vértices de $\cup_{i=1}^{MAX} C_i$ do conjunto de candidatos e os colocamos no conjunto de vértices visitados, obtendo o subproblema raiz $(V(G) \setminus \cup_{i=1}^{MAX} C_i, \Cup_{i=1}^{|I^\star|} C_i, I^\star)$, a partir do qual iniciaremos nossa busca. Para $k = MAX + 1, \ldots, n$, a inclusão do vértice v em W_k na linha 11 produz um subgrafo induzido $G_k + v$ de G tal que $\alpha(G_k + v) \leq MAX + 1$. Essa é a mesma situação da linha 3 do Algoritmo 10. Dessa forma, o teste da linha 11 responde a pergunta: *existe um conjunto independente*

Algoritmo 11: Descrição da sequência de subproblemas

```
1  Seja R um mapa de bits com os elementos de V
2  Seja W um mapa de bits inicialmente vazio
3  Determine valor inicial de MAX com Algoritmo 8
4  k ← coberturaVert(G, R, W, MAX)
5  enquanto k < n faça
6      Determine o menor vértice v de R
7      se α(G[W_k] ∩ N̄(v)) = MAX então
8          MAX ← MAX + 1
9      R ← R − v
10     W ← W + v
11     k ← k + 1
```

de tamanho $i_{max} + 1$*, no conjunto* $V(G_k) + v$*?*. O vértice v escolhido é o menor em R. Assim sendo, a sequência dos subproblemas tem os subproblemas iniciais definidos de acordo com a cobertura por cliques inicial e, em seguida, segue a numeração dos vértices. A introdução de critérios de eliminação de subproblemas altera essa sequência na forma descrita a seguir.

3.5 Eliminação de Subproblemas

O detalhamento do algoritmo *RD_MaxStab*, apresentado no Algoritmo 4, é feito através da especificação dos critérios de eliminação de subproblemas. Neste detalhamento, que é baseado no Algoritmo 7, usamos os métodos da interface da classe *BitMap* para manipulação dos conjuntos envolvidos. O teste da linha 11 do Algoritmo 11 é detalhado na função *Stab*() do Algoritmo 13.

Ao determinar $\alpha(G[W + v])$ temos duas possíveis situações:

- Se $\alpha(G[W + v]) = MAX$. Apenas incluímos v em W e prosseguimos para a próxima iteração.

- Se $\alpha(G[W + v]) = MAX + 1$. Seja $\hat{I} \subseteq W + v$ um conjunto independente de cardinalidade $\alpha(G[W + v])$. Expandimos este conjunto $\hat{I}$ para um conjunto independente maximal I de G, determinamos uma cobertura de cliques

$C_1, \ldots, C_k$ de $G[R]$ e retiramos todos os vértices de $\cup_{i=1}^{|I|-MAX-1} C_i$ de R e uma clique $C_v \subseteq R$ que contenha v e atualizamos o valor de MAX para $|I|$.

A retirada dos vértices de C_v do conjunto de candidatos é facilmente justificada pelo fato de um conjunto máximo de $G[W+v]$ conter v e assim, não pode conter mais nenhum vértice de C_v. A retirada dos vértices de $\uplus_{i=1}^{|I|-MAX-1} C_i$ é mais difícil de ser justificada e é explicada no Lema 1, mas uma maneira intuitiva de ver a validade desta remoção de vértices é que, devido a expansão de $\hat{I}$, podemos remover $|I| - |\hat{I}|$ cliques do conjunto de candidatos. Uma clique para cada vértice incluso em $\hat{I}$.

Algoritmo 12: *RD_MaxStab*

Entrada: grafo $G = (V, E)$.
Saída: $\alpha(G)$

```
1  Seja R um mapa de bits com os elementos de V
2  Seja W um mapa de bits inicialmente vazio
3  Determine o valor inicial de MAX com Algoritmo 8
4  k ← coberturaVert(G, R, W, MAX)
5  enquanto k < n faça
6      v ← R− > lsb()
7      I ← {v}
8      se STAB(W ∩ N̄(v), I, MAX) então
9          Determine I* tal que I ⊆ I* e I* é maximal em G
10         k ← cliqueMaximal(G, R, W, |I*| − MAX) MAX ←| I* |
11     senão
12         R− > toCompl(v)
13         W− > toSet(v)
14         k ← k + 1
15 retorna MAX
```

Observe que as operações realizadas em cada iteração do Algoritmo 12 são bem simples e tem complexidade de no máximo $O(n^2)$. Com a técnica de paralelismo de bits, estas operações podem ser feitas de maneira mais eficiente. Observe que, basicamente, as únicas operações realizadas pelo algoritmo *RD_MaxStab* são interseções de vizinhanças que se beneficiam do paralelismo de bits da maneira já examinada.

Algoritmo 13: Teste da linha 11 do Algoritmo 11

Entrada: conjunto de vértices R, conjunto independente I e limite lim.
Saída: verdadeiro, se existe um conjunto independentes em $G[R]$ de cardinalidade maior que lim; falso, em caso contrário

```
1  se lim = 0 e W ≠ ∅ então
2      retorna verdadeiro
3  se |R| < lim então
4      retorna falso
5  Sejam C_1, C_2, ... C_lim cliques de G[R]
6  W ← ∪_{i=0}^{lim} C_i
7  R ← R \ W
8  se W' = ∅ então retorna falso
9  enquanto R ≠ ∅ faça
10     v ← R-> lsb()
11     I ← I + v
12     se STAB(W ∩ N̄(v), I, lim − 1) então
13         retorna verdadeiro
14     I ← I − v
15     W-> toSet(v)
16     R-> toCompl(v)
17 retorna falso
```

Um ponto importante a ser destacado é que o algoritmo *RD_MaxStab* é aquele que melhor integra o paralelismo de bits a suas operações. Observe que não precisamos examinar os vértices do conjunto de candidatos segundo uma ordem pré-determinada. A não necessidade desta ordem permite o uso de mapas de bits sem o a sobrecarga de um vetor de números para representar uma ordem entre os vértices do conjunto de candidatos.

Para finalizar a apresentação do algoritmo, justificaremos a remoção dos elementos que ocorre na linha 11 do Algoritmo 12. Isto é feito com o lema a seguir.

Neste ponto vale ressaltar que as cliques removidas não tem nenhuma ligação com os vértices adicionados ao conjunto independente para torná-lo maximal. Poderíamos fazer desta maneira, fixando que as cliques devem cobrir os vértices inclusos no conjunto independente. Entretanto, isto pode piorar o desem-

penho do algoritmo por forçar mais operações e diminuir o número de vértices removidos em cada iteração.

Lema 1. *Seja W e R uma partição de $V(G)$. Seja $I \subseteq V(G)$ um conjunto independente maximal de G, tal que $\alpha(G[W]) = |I \cap W|$ e $C_1, \ldots, C_{|k|}$ uma cobertura de cliques de $G[R]$, então*

$$\alpha(G[W \cup \Cup_{i=1}^{|I \cap R|} C_i]) \leq |I|$$

Demonstração. Como $\alpha(G[W]) = |I \cap W|$ e $C_1, \ldots, C_k$ é uma cobertura de cliques de $G[R]$, temos que

$$\alpha(G[W \cup \Cup_{i=1}^{|I \cap R|} C_i]) \leq \alpha(G[W]) + \alpha(G[\Cup_{i=1}^{|I \cap R|} C_i]) \leq |I \cap W| + \alpha(G[\Cup_{i=1}^{|I \cap R|} C_i])$$

Observe agora que $\alpha(G[\Cup_{i=1}^{|I \cap R|} C_i]) \leq |I \cap R|$ uma vez que C_i é uma clique para todo i e logo um conjunto independente de $G[\Cup_{i=1}^{|I \cap R|} C_i]$ só pode conter um elemento de cada clique. Com o fato que $R \cap W = \emptyset$ o resultado segue. □

4 EXPERIMENTOS COMPUTACIONAIS

Neste capítulo apresentamos os resultados obtidos com a implementação realizada.

Inicialmente, comparamos nossa implementação do algoritmo descrito em (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011) com os resultados descritos no mesmo artigo. O objetivo desta comparação é comparamos o *BB_MaxClique* com a implementação do *RD_MaxStab* descrito no capítulo anterior a fim de analisar a eficiência deste e o impacto das diferentes implementações de mapas de bits.

No final analisamos os resultados obtidos e apontamos direções para uma continuidade do trabalho. As principais conclusões a destacar são as seguintes. No que diz respeito ao algoritmo *RD_MaxStab*, este é competitivo em relação ao algoritmo *BB_MaxClique* e apresenta um menor consumo de memória. No tocante as estruturas de dados, a tabela de dispersão apresenta uma economia de memória considerável, apesar de apresentar uma perda de eficiência.

4.1 Introdução

Os experimentos cujos resultados são apresentados e analisados neste capítulo foram realizados em um computador com arquitetura de 64 bits, memória principal de 8Gb e 2 microprocessadores intel *quad core*, instalado no LIA - Laboratório de Pesquisa em Ciência da Computação do Programa de Mestrado e Doutorado em Ciência da Computação da Universidade Federal do Ceará.

Como a arquitetura é um fator determinante para avaliar a performance dos programas, é necessário adotar uma maneira de normalizar os valores obtidos pela nossa implementação em relação com as demais na literatura. Adotamos a mesma metodologia utilizada em vários artigos na literatura qu apresentam resultados experimentais e análises desses resultados. Essa metodologia consiste em executar, no computador em que os experimentos foram realizados, um programa

de controle, denominado *dfmax*[1], com as instâncias $r100.5$, $r200.5$, $r300.5$, $r400.5$, $r500.5$. Os tempos de execução assim obtidos são listados na Tabela 1. De posse destes dados normalizamos os tempos em (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011), por um fator de correção obtido pela relação dos tempos nas duas máquinas. Considerando os dados na Tabela, este fator é de 1 1,66.

Um ponto importante a ser citado aqui é que, conforme descrito em (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011), o algoritmo *BB_MaxClique* determina uma clique máxima e não um conjunto independente máximo. O algoritmo *RD_MaxStab*, entretanto, é descrito determina um conjunto independente máximo. Resolvemos isto utilizando o complemento do grafo de entrada no algoritmo *RD_MaxStab*. Desta maneira, *RD_MaxStab* é utilizado para determinar a clique máxima e assim podemos compara-lo com *BB_MaxClique* e com a implementação descrita em (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011).

Outro ponto importante, que deve ser ressaltado é que reordenamos os vértices do grafo de entrada segundo a ordem descrita em (TOMITA; KAMEDA, 2007) de modo a melhorar o limite obtido em cada subproblema.

A seguinte menção pode ser encontrada em (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011):

> "An important effort has been made in optimizing all three algorithms. This might explain discrepancies between user times ... w.r.t. other results found elsewhere."

A fim de estabelecer uma base de comparação com os resultados apresentados em (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011), os tempos medidos, os tempos de leitura do grafo e reordenação dos vértices são desconsiderados, de *BB_MaxClique* presentes em (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011) corrigidos pelo fator médio 1,66 constam na Tabela 2.

Na Tabela 2, são comparados os resultados das duas implementações no tocante às instâncias do jogo de teste DIMACS. Este conjunto de instâncias, disponível em

[1] disponível em *ftp://dimacs.rutgers.edu/pub/dsj/clique/dfmax.c*

http://cs.hbg.psu.edu/txn131/clique.html, é utilizado em quase toda a literatura na área. Estes grafos contém estruturas e construções especiais para tornar difícil a resolução exata do problema CIM.

Instância	(SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011)	LIA	Fator
$r100.5$	0,00	0,00	-
$r200.5$	0,031	0,04	1,29
$r300.5$	0,234	0,39	1,67
$r400.5$	1,531	2,41	1,57
$r500.5$	5,766	9,18	1,59

Tabela 1: Valores usados para o cálculo do fator de ajuste de 1,61, considerando-se apenas as instâncias $r300.5$, $r400.5$ e $r500.5$.

4.2 Comparação *RD_MaxStab* x *BB_MaxClique*

Comparamos as implementações desenvolvidas dos algoritmos *RD_MaxStab* e *BB_MaxClique*. Inicialmente, utilizamos a estrutura de dados *Bit-Map* para comparar o desempenho dos dois algoritmos, não só no tocante ao tempo de execução, mas também quanto à quantidade de memória consumida pelos algoritmos. Estas comparações são apresentadas nas Tabelas 3 e 4. Na Tabela 3, temos o comparativo do tempo de execução, medido em segundos entre os dois algoritmos e na Tabela 4 temos o comparativo entre o consumo de memoria máximo durante a execução, ou seja, a maior quantidade de memória usada simultaneamente por cada algoritmo para armazenar os subproblemas e estruturas necessárias para a coloração. Observe que desconsideramos a memória consumida pelo grafo.

Com base nos experimentos apresentados nas Tabelas 4 e 3 vemos que o algoritmo *RD_MaxStab* é competitivo, no tocante ao tempo, em relação ao *BB_MaxClique*, sendo em média mais rápido e tem um consumo de memória, em média 71 vezes menos.

Esta observação nós leva a crer que o algoritmo *RD_MaxStab* deve se comportar melhor que o algoritmo *BB_MaxClique* em casos de uso prático, onde os grafos não apresentam estrutura especial e são em geral muito grandes. Para

Instância	Tempo		Número de Nós		Razão	
	SDRJ	*BitMap*	*SDRJ*	*BitMap*	Tempo	Nós
brock200_2	–	0,0169	4004	3822	–	1,048
brock200_4	0,12948	0,3357	64676	57744	2,593	1,120
brock400_2	350,62188	1008,3776	66381296	71987904	2,876	0,922
brock400_4	332,31042	876,3663	66669902	60894508	2,637	1,095
c-fat200-1	–	0,0002	216	220	–	0,982
c-fat200-2	–	0,0006	241	244	–	0,988
c-fat200-5	–	0,0020	308	312	–	0,987
c-fat500-10	0,05146	0,0130	744	748	0,254	0,995
c-fat500-1	–	0,0008	520	524	–	0,992
c-fat500-2	–	0,0017	544	548	–	0,993
c-fat500-5	0,02656	0,0049	620	624	0,185	0,994
hamming8-4	0,05146	0,0581	289	7834	1,130	0,037
keller4	0,02656	0,0598	12105	13446	2,254	0,900
MANN_a27	0,90802	2,6795	38253	44474	2,951	0,860
MANN_a45	402,70604	1037,6433	2953135	3188208	2,577	0,926
p_hat300-1	–	0,0068	1982	1982	–	1,000
p_hat300-2	0,05146	0,0593	6226	6693	1,153	0,930
p_hat300-3	3,19052	8,4785	590226	575206	2,657	1,026
p_hat700-1	0,12948	0,2269	30253	30113	1,753	1,005
p_hat700-2	9,07688	18,171	714868	809791	2,002	0,883
san200_0.7_2	0,02656	0,0091	1346	1377	0,343	0,977
san200_0.9_1	0,49136	0,8436	110813	65154	1,717	1,701
san200_0.9_2	0,31208	0,2236	62766	12068	0,717	5,201
san200_0.9_3	0,07802	0,2151	13206	16608	2,757	0,795

Tabela 2: Comparação dos número de subproblemas gerados e tempo de execução entre *BB_MaxClique* de (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011), indicado por *SDRJ*, e da implementação com *BitMap*, indicado por *BitMap*. Os tempos de execução de (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011) estão ajustados pelo fator 1,61. A razão apresentada nas duas últimas colunas correspondem à medida (número de subproblemas ou tempo) para a implementação usando *BitMap* dividida pela mesmaem (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011). Fatores médios apurados são: para o número de subproblemas, 0,89, e para o tempo de execução, 1,66.

examinar nossa suspeita, observamos um problema de programação inteira descrita em (BEN-DOR; SHAMIR; YAKHINI, 1999).

Não entraremos em detalhes sobre o problema descrito em (BEN-DOR; SHAMIR; YAKHINI, 1999) e da sua solução aqui. Basta saber que um dos passos da solução demanda a determinação de um conjunto independente do grafo de entrada para o problema.

Instância	Número de Nós			Tempo		
	BB	*RD*	Razão	*BB*	*RD*	Razão
brock200_2	3822	5505	1,44	0,016	0,028	1,750
brock200_4	57744	45233	0,78	0,33	0,306	0,927
brock400_2	71987904	127751663	1,77	994,6026	1627,4	1,636
brock400_4	60894508	66942184	1,10	864,7711	862,7	0,998
c-fat200-1	220	4	0,02	0,0001	0,00005	0,500
c-fat200-2	244	1	0,00	0,0004	0,000007	0,018
c-fat200-5	312	27	0,09	0,0014	0,0003	0,214
c-fat500-1	524	1	0,00	0,0004	0,00001	0,025
c-fat500-2	548	1	0,00	0,0007	0,000007	0,010
c-fat500-5	624	1	0,00	0,0025	0,00001	0,004
c-fat500-10	748	1	0,00	0,0082	0,00001	0,001
hamming8-4	7834	6870	0,88	0,057	0,065	1,140
keller4	13446	20522	1,53	0,0587	0,088	1,499
MANN_a27	44474	44680	1,00	2,5154	3,566	1,418
MANN_a45	3188208	3523403	1,11	994,6889	1428,9	1,437
p_hat300-1	1982	1227	0,62	0,006889	0,0066	0,958
p_hat300-2	6693	2857	0,43	0,0587	0,045	0,767
p_hat300-3	575206	156797	0,27	8,3749	3,318	0,396
p_hat700-1	30113	16541	0,55	0,2195	0,212	0,966
p_hat700-2	809791	281187	0,35	17,6278	11,91	0,676
san200_0.7_2	1377	338	0,25	0,009	0,0027	0,300
san200_0.9_1	65154	443	0,01	0,846	0,005	0,006
san200_0.9_2	12068	325	0,03	0,2193	0,0034	0,016
san200_0.9_3	16608	2610	0,16	0,2195	0,045	0,205

Tabela 3: Comparativo entre os algoritmos *BB_MaxClique* e *RD_MaxStab* . Os tempos estão medidos em segundos e a coluna razão simboliza a divisão do tempo/numero de nós de *RD_MaxStab* por *BB_MaxClique*. Os fatores médios apurados são de 0,5 para o número de nós e 0,77 para o tempo de execução.

Utilizamos então as instâncias descritas em (BEN-DOR; SHAMIR; YAKHINI, 1999) para examinar o desempenho prático do algoritmo *RD_MaxStab* e *BB_MaxClique*.

Apresentamos a comparação do desempenho destes dois algoritmos nas Tabelas 5 e 7. Na Tabela 5, temos o comparativo do tempo de execução entre as duas implementações, onde vemos que *RD_MaxStab* é muitas vezes mais rápido que *BB_MaxClique*. Na Tabela 7 temos o comparativo da memória utilizada pelas duas implementações e utilizada para representar o grafo de entrada. Nesta tabela observamos que o consumo de memória de *RD_MaxStab* é muito inferior ao con-

Instância	Grafo	*BB*	*RD*	$\frac{Grafo}{BB}$	$\frac{Grafo}{RD}$	$\frac{BB}{RD}$
brock200_2	7,81	31,4	1,4	0,179	4,019	22,429
brock200_4	7,81	57,5	1,8	0,230	7,360	31,944
brock400_2	25,00	330,9	4,2	0,168	13,236	78,786
brock400_4	25,00	279,7	4,2	0,168	11,188	66,595
c-fat200-1	7,81	10,4	0,4	0,051	1,331	26,000
c-fat200-2	7,81	11,1	0,2	0,026	1,421	55,500
c-fat200-5	7,81	36,7	0,4	0,051	4,698	91,750
c-fat500-1	35,16	25,1	0,3	0,009	0,714	83,667
c-fat500-2	35,16	26,3	0,3	0,009	0,748	87,667
c-fat500-5	35,16	47,4	0,3	0,009	1,348	158,000
c-fat500-10	35,16	152,8	0,3	0,009	4,346	509,333
hamming8-4	10,00	31,2	1	0,100	3,120	31,200
keller4	5,34	38,8	1,1	0,206	7,261	35,273
MANN_a27	20,67	236	17	0,822	11,416	13,882
MANN_a45	145,55	1743	105,7	0,726	11,976	16,490
p_hat300-1	14,06	32,4	0,7	0,050	2,304	46,286
p_hat300-2	14,06	54,7	3,2	0,228	3,890	17,094
p_hat300-3	14,06	132,8	4,5	0,320	9,444	29,511
p_hat700-1	65,63	288,8	2,8	0,043	4,401	103,143
p_hat700-2	65,63	432,1	9,7	0,148	6,584	44,546
san200_0.7_2	7,81	15,4	1,9	0,243	1,971	8,105
san200_0.9_1	7,81	23,3	6,7	0,858	2,982	3,478
san200_0.9_2	7,81	25	6,1	0,781	3,200	4,098
san200_0.9_3	7,81	21,7	4,7	0,602	2,778	4,617

Tabela 4: Comparativo entre os algoritmos *BB_MaxClique* e *RD_MaxStab* . A memória está medida em kilobytes. O algoritmo *BB_MaxClique* utiliza em média 71 vezes mais memória que o algoritmo *RD_MaxStab*.

sumo de memória de *BB_MaxClique*, entretanto, a memória consumida pelo grafo é maior que o consumo de memória de ambos os algoritmos.

4.3 Efeito da Tabela de Dispersão no Consumo de Memória

Como uma das principais vantagens do algoritmo *RD_MaxStab* é a economia de memória, isso nos leva a necessidade de diminuir a quantidade de memória necessário para representar o grafo, isso é obtido utilizando a estrutura *PHashBitMap* apresentada anteriormente.

Como a grande maioria dos grafos apresentados em (BEN-DOR; SHA-

MIR; YAKHINI, 1999) são esparsos, o tamanho do nó tem grande influência sobre o tamanho da tabela como não é difícil de se perceber, por isso estamos esta estrutura com dois tamanhos para os nós, 8 e 64 bits.

Como o *PHashBitMap* é uma estrutura estática, não permitindo exclusões e inclusões, o algoritmo *RD_MaxStab* não pode utilizá-lo internamente para representar os conjuntos de candidatos. Neste ponto nossa implementação mostra sua maior qualidade, pois através da interface padrão e dos iteradores podemos utilizar a estrutura *ArrayBitMap* para representar os conjuntos de candidatos nos subproblemas de *RD_MaxStab* sem nenhuma modificação no código da enumeração.

Podemos observar que a economia de memória com a utilização da tabela de dispersão para representar o grafo é muito grande, chegando a tabela a usar apenas 10% da memória utilizada pelo vetor de bits. Entretanto, o tempo de execução aumenta em um fator de quase 3 vezes. Desta forma, obtivemos uma economia de memória considerável em detrimento ao tempo de execução.

4.4 Efeito da Estratificação

Para finalizar nossos experimentos, uma última análise a ser feita é a utilidade do sumário na implementação. Com este objetivo, refazemos os experimentos com os dados da aplicação (BEN-DOR; SHAMIR; YAKHINI, 1999) com a estrutura de sumário. Os testes são realizados com as classes concretas *StratifiedArrayBitMap*, Tabela 8 e *StratifiedPHashBitMap*, Tabela 9. utilizamos apenas um nível de sumário em cada experimento.

Instância	Número de Nós			Tempo		
	BB	*RD*	$\frac{RD}{BB}$	*BB*	*RD*	$\frac{RD}{BB}$
1AX8.Sigma0.01.D050	1023	217	0,21	0,0023	0,0015	0,65
1AX8.Sigma0.01.D070	1066	162	0,15	0,0031	0,0015	0,48
1AX8.Sigma0.01.D100	1111	132	0,12	0,0045	0,0013	0,29
1BPM.Sigma0.01.D050	3711	518	0,14	0,0165	0,0107	0,65
1BPM.Sigma0.01.D070	3750	765	0,20	0,0207	0,017	0,82
1BPM.Sigma0.01.D100	3806	236	0,06	0,0323	0,0076	0,24
1F39.Sigma0.01.D050	1565	173	0,11	0,0039	0,0017	0,44
1F39.Sigma0.01.D070	1553	281	0,18	0,0054	0,0035	0,65
1F39.Sigma0.01.D100	1620	99	0,06	0,0081	0,0015	0,19
1HOE.Sigma0.01.D050	578	131	0,23	0,001	0,0007	0,70
1HOE.Sigma0.01.D070	591	130	0,22	0,0014	0,0008	0,57
1HOE.Sigma0.01.D100	615	99	0,16	0,0021	0,0007	0,33
1HQQ.Sigma0.01.D050	4002	481	0,12	0,0181	0,0107	0,59
1HQQ.Sigma0.01.D070	4047	664	0,16	0,0248	0,0174	0,70
1HQQ.Sigma0.01.D100	4133	192	0,05	0,0369	0,0069	0,19
1KDH.Sigma0.01.D050	2882	331	0,11	0,0105	0,0057	0,54
1KDH.Sigma0.01.D070	2913	320	0,11	0,0141	0,0066	0,47
1KDH.Sigma0.01.D100	2981	82	0,03	0,0203	0,002	0,10
1LFB.Sigma0.01.D050	661	133	0,20	0,0012	0,0007	0,58
1LFB.Sigma0.01.D070	707	113	0,16	0,0017	0,0008	0,47
1LFB.Sigma0.01.D100	753	71	0,09	0,0026	0,0005	0,19
1MQQ.Sigma0.01.D050	5712	846	0,15	0,0345	0,0241	0,70
1MQQ.Sigma0.01.D070	5809	1163	0,20	0,0462	0,0391	0,85
1MQQ.Sigma0.01.D100	5827	150	0,03	0,0681	0,0065	0,10
1PHT.Sigma0.01.D050	849	174	0,20	0,0019	0,0017	0,89
1PHT.Sigma0.01.D070	874	37	0,04	0,0026	0,0014	0,54
1PHT.Sigma0.01.D100	977	202	0,21	0,0039	0,0003	0,08
1POA.Sigma0.01.D050	931	135	0,15	0,002	0,0013	0,65
1POA.Sigma0.01.D070	949	85	0,09	0,0026	0,0011	0,42
1POA.Sigma0.01.D100	1028	85	0,08	0,0041	0,001	0,24
1PTQ.Sigma0.01.D050	414	80	0,19	0,0007	0,0004	0,57
1PTQ.Sigma0.01.D070	432	69	0,16	0,0009	0,0004	0,44
1PTQ.Sigma0.01.D100	496	87	0,18	0,0014	0,0006	0,43

Tabela 5: Comparativo entre os algoritmos *BB_MaxClique* (coluna *BB*) e *RD_MaxStab* (coluna *BB*). Os tempos estão medidos em segundos e a coluna $\frac{RD}{BB}$ simboliza a divisão do tempo/numero de nós de *BB_MaxStab* por *BB_MaxClique*. Os fatores médios apurados são 0,13 para o número de nós e 0,47 para o tempo.

Instância	Grafo	*BB*	*RD*	$\frac{Grafo}{BB}$	$\frac{Grafo}{RD}$	$\frac{BB}{RD}$
1AX8.Sigma0.01.D050	133,21	48,9	2,98	0,022	0,367	16,409
1AX8.Sigma0.01.D070	133,21	49,3	2,32	0,017	0,370	21,250
1AX8.Sigma0.01.D100	133,21	51,2	6,91	0,052	0,384	7,410
1BPM.Sigma0.01.D050	1692,1	177,3	8,88	0,005	0,105	19,966
1BPM.Sigma0.01.D070	1692,1	178,3	10,9	0,006	0,105	16,358
1BPM.Sigma0.01.D100	1692,1	183,4	20,7	0,012	0,108	8,860
1F39.Sigma0.01.D050	299,61	74,4	4,1	0,014	0,248	18,146
1F39.Sigma0.01.D070	299,61	74,9	5	0,017	0,250	14,980
1F39.Sigma0.01.D100	299,61	77,2	9,1	0,030	0,258	8,484
1HOE.Sigma0.01.D050	43,59	27,4	1,8	0,041	0,629	15,222
1HOE.Sigma0.01.D070	43,59	27,6	2,2	0,050	0,633	12,545
1HOE.Sigma0.01.D100	43,59	29	4,4	0,101	0,665	6,591
1HQQ.Sigma0.01.D050	1941,19	190	9,4	0,005	0,098	20,213
1HQQ.Sigma0.01.D070	1941,19	191,5	11,5	0,006	0,099	16,652
1HQQ.Sigma0.01.D100	1941,19	196,4	21	0,011	0,101	9,352
1KDH.Sigma0.01.D050	1022,78	137,6	7,1	0,007	0,135	19,380
1KDH.Sigma0.01.D070	1022,78	138,8	9,4	0,009	0,136	14,766
1KDH.Sigma0.01.D100	1022,78	142,5	16,4	0,016	0,139	8,689
1LFB.Sigma0.01.D050	60,09	31,4	2	0,033	0,523	15,700
1LFB.Sigma0.01.D070	60,09	31,8	2,8	0,047	0,529	11,357
1LFB.Sigma0.01.D100	60,09	33,2	5	0,083	0,553	6,640
1MQQ.Sigma0.01.D050	3994,45	273,9	13,2	0,003	0,069	20,750
1MQQ.Sigma0.01.D070	3994,45	275,4	16,2	0,004	0,069	17,000
1MQQ.Sigma0.01.D100	3994,45	284,5	33,8	0,008	0,071	8,417
1PHT.Sigma0.01.D050	88,7	39,5	2,3	0,026	0,445	17,174
1PHT.Sigma0.01.D070	88,7	40,2	3,4	0,038	0,453	11,824
1PHT.Sigma0.01.D100	88,7	42,3	6,8	0,077	0,477	6,221
1POA.Sigma0.01.D050	114,25	44,5	2,5	0,022	0,389	17,800
1POA.Sigma0.01.D070	114,25	45,2	3,8	0,033	0,396	11,895
1POA.Sigma0.01.D100	114,25	46,6	6,3	0,055	0,408	7,397
1PTQ.Sigma0.01.D050	25,13	19,9	1	0,040	0,792	19,900
1PTQ.Sigma0.01.D070	25,13	20,3	2,1	0,084	0,808	9,667
1PTQ.Sigma0.01.D100	25,13	21,1	3,4	0,135	0,840	6,206

Tabela 6: Comparativo entre os algoritmos *BB_MaxClique* (coluna *BB*) e *RD_MaxStab* (coluna *RD*) . A memória está medida em bytes e as colunas $\frac{Grafo}{BB}$ e $\frac{Grafo}{RB}$ representam, respectivamente, a divisão da memória utilizada pelo algoritmo pela memória usada pelo grafo. A última coluna representa a divisão da memória consumida pelo *BB_MaxClique* sobre a memória utilizada pelo *RD_MaxStab*. Os fatores médios apurados são de 0,05 para a coluna $\frac{Grafo}{BB}$, 0,33 para a coluna $\frac{Grafo}{RD}$ e 12,4 para a coluna $\frac{BB}{RD}$.

Instância	Memória Grafo			Tempo		
	Array	*PHash8*	*PHash64*	*Array*	*PHash8*	*PHash64*
1AX8.Sigma0.01.D050	133,21	107,60	89,45	0,0015	0,0795	0,0114
1AX8.Sigma0.01.D070	133,21	124,54	89,02	0,0015	0,0737	0,0108
1AX8.Sigma0.01.D100	133,21	143,36	95,47	0,0013	0,0564	0,0082
1BPM.Sigma0.01.D050	1692,1	459,96	411,20	0,0107	0,8056	0,1102
1BPM.Sigma0.01.D070	1692,1	543,85	428,13	0,017	1,2187	0,1683
1BPM.Sigma0.01.D100	1692,1	633,85	473,25	0,0076	0,4900	0,0669
1F39.Sigma0.01.D050	299,61	166,77	132,48	0,0017	0,1041	0,0146
1F39.Sigma0.01.D070	299,61	206,42	155,50	0,0035	0,1983	0,0277
1F39.Sigma0.01.D100	299,61	234,50	165,50	0,0015	0,0789	0,0111
1HOE.Sigma0.01.D050	43,59	60,37	41,88	0,0007	0,0247	0,0037
1HOE.Sigma0.01.D070	43,59	67,84	44,24	0,0008	0,0301	0,0046
1HOE.Sigma0.01.D100	43,59	76,60	45,05	0,0007	0,0237	0,0037
1HQQ.Sigma0.01.D050	1941,19	484,17	436,59	0,0107	0,8162	0,1113
1HQQ.Sigma0.01.D070	1941,19	583,84	503,54	0,0174	1,2796	0,1740
1HQQ.Sigma0.01.D100	1941,19	680,36	521,90	0,0069	0,4717	0,0645
1KDH.Sigma0.01.D050	1022,78	326,23	271,51	0,0057	0,4011	0,0552
1KDH.Sigma0.01.D070	1022,78	379,61	299,23	0,0066	0,4597	0,0636
1KDH.Sigma0.01.D100	1022,78	434,80	314,94	0,002	0,1212	0,0168
1LFB.Sigma0.01.D050	60,09	67,07	49,62	0,0007	0,0304	0,0046
1LFB.Sigma0.01.D070	60,09	75,66	52,49	0,0008	0,0313	0,0049
1LFB.Sigma0.01.D100	60,09	84,71	55,43	0,0005	0,0175	0,0028
1MQQ.Sigma0.01.D050	3994,45	731,71	629,95	0,0241	1,9506	0,2600
1MQQ.Sigma0.01.D070	3994,45	872,12	700,04	0,0391	3,0695	0,4127
1MQQ.Sigma0.01.D100	3994,45	1,010,65	755,01	0,0065	0,4775	0,0635
1PHT.Sigma0.01.D050	88,7	102,33	70,75	0,0017	0,0784	0,0115
1PHT.Sigma0.01.D070	88,7	116,93	72,33	0,0014	0,0667	0,0099
1PHT.Sigma0.01.D100	88,7	130,71	77,11	0,0003	0,0096	0,0015
1POA.Sigma0.01.D050	114,25	98,64	78,76	0,0013	0,0618	0,0091
1POA.Sigma0.01.D070	114,25	111,19	79,92	0,0011	0,0578	0,0085
1POA.Sigma0.01.D100	114,25	133,20	90,55	0,001	0,0404	0,0061
1PTQ.Sigma0.01.D050	25,13	41,31	27,73	0,0004	0,0120	0,0019
1PTQ.Sigma0.01.D070	25,13	45,67	29,04	0,0004	0,0123	0,0020
1PTQ.Sigma0.01.D100	25,13	53,60	33,54	0,0006	0,0164	0,0026

Tabela 7: Comparativo entre as estruturas de *ArrayBitMap* e *PHashBitMap*, utilizados no algoritmo *RD_MaxStab* . A estrutura *PHashBitMap* foi testada com dois tamanhos diferentes de palavra, 8 bits e 64 bits.

Instância	Memória Grafo			Tempo		
	Array	*StArr8*	*StArr64*	*Array*	*StArr8*	*StArr64*
1AX8.Sigma0.01.D050	133,21	145,99	132,56	0,0015	0,0069	0,0015
1AX8.Sigma0.01.D070	133,21	145,99	132,56	0,0015	0,0066	0,0015
1AX8.Sigma0.01.D100	133,21	145,99	132,56	0,0013	0,0057	0,0012
1BPM.Sigma0.01.D050	1692,1	1879,36	1699,45	0,0107	0,0599	0,0107
1BPM.Sigma0.01.D070	1692,1	1879,36	1699,45	0,017	0,0998	0,0170
1BPM.Sigma0.01.D100	1692,1	1879,36	1699,45	0,0076	0,0415	0,0098
1F39.Sigma0.01.D050	299,61	335,14	303,73	0,0017	0,0088	0,0017
1F39.Sigma0.01.D070	299,61	335,14	303,73	0,0035	0,0177	0,0033
1F39.Sigma0.01.D100	299,61	335,14	303,73	0,0015	0,0085	0,0032
1HOE.Sigma0.01.D050	43,59	47,12	42,98	0,0007	0,0024	0,0006
1HOE.Sigma0.01.D070	43,59	47,12	42,98	0,0008	0,0031	0,0008
1HOE.Sigma0.01.D100	43,59	47,12	42,98	0,0007	0,0027	0,0006
1HQQ.Sigma0.01.D050	1941,19	2166,99	1959,30	0,0107	0,0668	0,0107
1HQQ.Sigma0.01.D070	1941,19	2166,99	1959,30	0,0174	0,1072	0,0173
1HQQ.Sigma0.01.D100	1941,19	2166,99	1959,30	0,0069	0,0422	0,0069
1KDH.Sigma0.01.D050	1022,78	1134,56	1026,43	0,0057	0,0336	0,0057
1KDH.Sigma0.01.D070	1022,78	1134,56	1026,43	0,0066	0,0387	0,0066
1KDH.Sigma0.01.D100	1022,78	1134,56	1026,43	0,002	0,0114	0,0020
1LFB.Sigma0.01.D050	60,09	61,44	55,96	0,0007	0,0029	0,0007
1LFB.Sigma0.01.D070	60,09	61,44	55,96	0,0008	0,0030	0,0007
1LFB.Sigma0.01.D100	60,09	61,44	55,96	0,0005	0,0019	0,0005
1MQQ.Sigma0.01.D050	3994,45	4476,51	4045,62	0,0241	0,1547	0,0245
1MQQ.Sigma0.01.D070	3994,45	4476,51	4045,62	0,0391	0,2466	0,0393
1MQQ.Sigma0.01.D100	3994,45	4476,51	4045,62	0,0065	0,0409	0,0065
1PHT.Sigma0.01.D050	88,7	96,66	87,88	0,0017	0,0072	0,0017
1PHT.Sigma0.01.D070	88,7	96,66	87,88	0,0014	0,0060	0,0014
1PHT.Sigma0.01.D100	88,7	96,66	87,88	0,0003	0,0010	0,0002
1POA.Sigma0.01.D050	114,25	121,87	110,72	0,0013	0,0058	0,0012
1POA.Sigma0.01.D070	114,25	121,87	110,72	0,0011	0,0054	0,0012
1POA.Sigma0.01.D100	114,25	121,87	110,72	0,001	0,0041	0,0009
1PTQ.Sigma0.01.D050	25,13	25,33	23,19	0,0004	0,0012	0,0003
1PTQ.Sigma0.01.D070	25,13	25,33	23,19	0,0004	0,0013	0,0003
1PTQ.Sigma0.01.D100	25,13	25,33	23,19	0,0006	0,0020	0,0005

Tabela 8: Comparativo entre as estruturas de *ArrayBitMap* e *StratifiedArrayBitMap*, utilizados no algoritmo *RD_MaxStab* . A estrutura *StratifiedArrayBitMap* foi testada com dois tamanhos diferentes de palavra, 8 bits e 64 bits.

Instância	Memória Grafo			Tempo		
	Array	*StPHas8*	*StPHas64*	*Array*	*StPHas8*	*StPHas64*
1AX8.Sigma0.01.D050	133,21	43,46	53	0,0015	0,0175	0,0055
1AX8.Sigma0.01.D070	133,21	46,85	52,77	0,0015	0,0178	0,0053
1AX8.Sigma0.01.D100	133,21	50,61	56,19	0,0013	0,0157	0,0047
1BPM.Sigma0.01.D050	1692,1	344,37	259,87	0,0107	0,1347	0,0290
1BPM.Sigma0.01.D070	1692,1	361,15	268,83	0,017	0,2003	0,0471
1BPM.Sigma0.01.D100	1692,1	379,15	292,72	0,0076	0,0890	0,0207
1F39.Sigma0.01.D050	299,61	90,88	81,77	0,0017	0,0192	0,0052
1F39.Sigma0.01.D070	299,61	98,81	93,96	0,0035	0,0406	0,0114
1F39.Sigma0.01.D100	299,61	104,42	99,25	0,0015	0,0174	0,0047
1HOE.Sigma0.01.D050	43,59	24,28	25,32	0,0007	0,0077	0,0023
1HOE.Sigma0.01.D070	43,59	25,77	26,56	0,0008	0,0098	0,0031
1HOE.Sigma0.01.D100	43,59	27,53	26,99	0,0007	0,0085	0,0024
1HQQ.Sigma0.01.D050	1941,19	367,98	276,45	0,0107	0,1233	0,0277
1HQQ.Sigma0.01.D070	1941,19	387,92	311,89	0,0174	0,2057	0,0480
1HQQ.Sigma0.01.D100	1941,19	407,22	321,61	0,0069	0,0812	0,0193
1KDH.Sigma0.01.D050	1022,78	260,91	176,44	0,0057	0,0645	0,0154
1KDH.Sigma0.01.D070	1022,78	271,58	191,11	0,0066	0,0770	0,0185
1KDH.Sigma0.01.D100	1022,78	282,62	199,43	0,002	0,0231	0,0055
1LFB.Sigma0.01.D050	60,09	27,43	29,88	0,0007	0,0083	0,0027
1LFB.Sigma0.01.D070	60,09	29,15	31,4	0,0008	0,0094	0,0031
1LFB.Sigma0.01.D100	60,09	30,96	32,96	0,0005	0,0061	0,0019
1MQQ.Sigma0.01.D050	3994,45	891,97	443,16	0,0241	0,2772	0,0563
1MQQ.Sigma0.01.D070	3994,45	920,05	480,26	0,0391	0,4595	0,0956
1MQQ.Sigma0.01.D100	3994,45	947,76	509,36	0,0065	0,0809	0,0153
1PHT.Sigma0.01.D050	88,7	38,21	42,02	0,0017	0,0214	0,0066
1PHT.Sigma0.01.D070	88,7	41,13	42,86	0,0014	0,0192	0,0056
1PHT.Sigma0.01.D100	88,7	43,88	45,39	0,0003	0,0032	0,0009
1POA.Sigma0.01.D050	114,25	39,72	46,84	0,0013	0,0146	0,0046
1POA.Sigma0.01.D070	114,25	42,23	47,46	0,0011	0,0140	0,0041
1POA.Sigma0.01.D100	114,25	46,63	53,08	0,001	0,0113	0,0034
1PTQ.Sigma0.01.D050	25,13	13,92	16,55	0,0004	0,0041	0,0013
1PTQ.Sigma0.01.D070	25,13	14,79	17,25	0,0004	0,0047	0,0015
1PTQ.Sigma0.01.D100	25,13	16,37	19,63	0,0006	0,0071	0,0022

Tabela 9: Comparativo entre as estruturas de *ArrayBitMap* e *StratifiedPHashBitMap*, utilizados no algoritmo *RD_MaxStab*. A estrutura *StratifiedPHashBitMap* foi testada com dois tamanhos diferentes de palavra, 8 bits e 64 bits.

5 CONCLUSÃO

A implementação para mapas de bits se mostrou competitiva nos experimentos realizados e flexível para proporcionar as mudanças na estrutura de dados adjacente sem modificar o código da aplicação que a utiliza, no nosso caso um algoritmo para o problema de conjunto independente máximo.

No tocante ao método das bonecas russas, a eficiência deste com a poda por cobertura por clique foi uma grata surpresa, em especial pela grande economia de memoria que este método proporciona.

5.1 Trabalhos Futuros

No final do trabalho, como de costume, apresentamos algumas direções para dar continuidade ao trabalho desenvolvido e obter novos resultados ou melhorar os resultados já mostrados.

Nas classe que implementam mapas de bits, podemos criar uma classe com implementações específicas das operações sobre conjuntos com cada classe que implementa *BitMap*. Com o intuito de fazer uso da estrutura específica de cada estrutura para acelerar estas operações.

No tocante aos algoritmos, uma direção a ser examinada é a adição do critério de recoloração do *MCS* (TOMITA; AKUTSU; MATSUNAGA, 2009) aos algoritmos *RD_MaxStab* e *BB_MaxClique*.

No tocante a implementação e as estruturas de dados a direção mais natural a ser examinada é a adição de paralelismo, desta vez, não apenas de bits, nas operações de ramificação e colorações dos subproblemas.

Nesta linha, o algoritmo *RD_MaxStab* parece ter muito a ganha com este paralelismo em dois níveis de bits e execução, devido aos subproblemas serem mais complexos, levam mais tempo para serem examinados.

REFERÊNCIAS

AKKOYUNLY, E. The Enumeration of Maxial Cliques of Large Graphs. *SIAM Journal on Computing*, v. 2, p. 1–6, 1973.

AURORA, S. et al. On the intractability of Approximantion Problems. *Preliminary Draft, University of Rochester*, 1992.

AURORA, S.; SAFRA, S. Approximating the Maximum Clique is NP-Complete. *Preliminary Draft, University of Rochester*, 1992.

BAEZA-YATES, R.; GONNET, G. A new approach to text searching. *Journal Association Computation Machine*, v. 35(10), p. 74–82, 1992.

BELAZZOUGUI, D.; BOTELHO, F.; DITZFELBINGER, M. Hash, displace, and compress. *Lecture Notes in Computer Scienc*, v. 5757, p. 682–693, 2009.

BEN-DOR, A.; SHAMIR, R.; YAKHINI, Z. Clustering Gene Expression Patterns. *Journal of Computational Biology*, v. 6, p. 281–297, 1999.

BOMZE, I.; BUDINICH, M.; M., P. *Handbook of Combinatorial Optimization*. [S.l.]: Kluwer Academic Publishers Dordrecht, 1999.

BRON, C.; KERBOSCH, J. Finding All Cliques of an Undirected Graph. *Commnucations of ACM*, v. 16, p. 575–577, 1973.

CORMEN, L.; RIVEST; STEIN, C. *Introduction to Algorithms*, 2º. ed, 2003.

LEISERSON, H.; PROKOP, K.; RANDALL. Using sequences to index a 1 in a computer word. *Manuscrito disponível em: http://supertech.csail.mit/papers/debrujin.pdf*, 1998.

PAPADIMITRIOU, C.; YANNAKAKIS, M. The Clique Problem for Planar Graphs. *Information Procedures Letters*, v. 13, 1981.

PARDALOS, P.; XUE, J. The maximum clique problem. *Journal of Global Optimization*, v. 4, p. 301–328, 1994.

SEGUNDO, P. S.; RODRÍGUEZ-LOSADA, D.; JIMÉNEZ, A. An exact bit-parallel algorithm for the maximum clique problem. *Computers and Operations Research*, v. 38, p. 571–581, 2011.

SEGUNDO, P. S.; RODRIGUEZ-LOSADA, D.; ROSSI, C. Recent Developments in Bit-Parallel Algorithms. *Tools in Artificial Intelligence*, p. 978–953, 2007.

SMRIGLIO, S. et al. Linear Relaxations for the Maximum Stable Set Problem: a computacional study. *http://www.euro-online.org*, 2008.

TOMITA, E.; AKUTSU, T.; MATSUNAGA, T. Efficient Algorithms for Finding Maximum and Maximal Cliques: Effective Tools for Bioinformatics. *Biomedical Engineering, Trends, Researches and Technologies*, v. 32, p. 625–641, 2009.

TOMITA, E.; KAMEDA, T. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global Optimization*, v. 37, p. 95–111, 2007.

VASKELAINEN, V. Russian Dolls Search Algorithms For Discrete Optimization Problems. *Alto University*, jun 2010.

VERFAILLIE, G.;LEMA, M.;SCHIEX, T. Russian Doll Search for Solving Constraint Optimization Problems. *AAAI*, v. 1, p.181–187, 1996.

DEMENTIEV, R.; KETTNER, L.; MEHNERT, J.; SANDERS, P. Engineering a sorted list data strucuture for 32 bits keys. *In Proceedings of the Sixth Workshop on Algorithm Enginnering and Experiments and the First WorkShop on Analytic Algorithmcs and Combinatorics*, p.142–155, 2004.

CORRÊA, R. and DONNE, D.D. and LECUN, B. and MAUTOR, T. MICHELON, P. The Russian Dolls Method and The Maximum Stable Set Problem. *Artigo em preparação. Este trabalho é fruto do projeto de cooperação cienífica STAB no Programa STIC/AmSud*, 2013.

ÖSTERG{AARD, P. R. J. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, v. 20, 2002.

APÊNDICE A – ALGUMAS DEFINIÇÕES E NOTAÇÃO

A.1 Teoria dos Grafos

Um *grafo* G é um par ordenado (V, E) em que $V = [n] = \{1, 2, \ldots, n\}, n \geq 1$, e E é um subconjunto de pares não ordenados de vértices. Os elementos de V são denominados *vértices* e os elementos de E são denominados de *arestas*. Uma aresta $(v, u) \in E$ é denotada simplesmente por uv. O conjunto de vértices pode também ser denotado por $V(G)$ e seu conjunto de arestas por $E(G)$. Se $E(G) = \{uv \mid u, v \in V, u \neq v\}$, dizemos que G é completo.

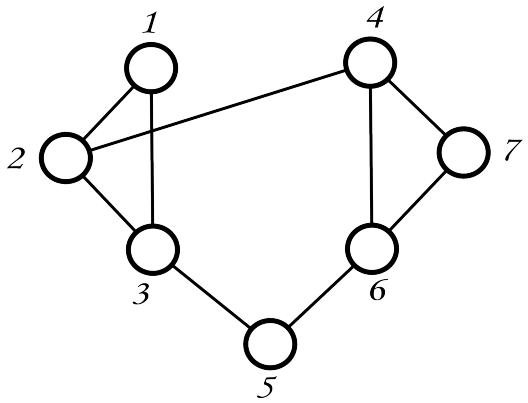


Figura 10: Grafo G, onde $V(G)$ = {1,2,3,4,5,6,7} e $E(G)$ = {(1,2), (2,1), (1,3), (3,1), (2,3), (3,2), (3,5), (5,3), (2,4), (4,2), (4,6), (6,4), (5,6), (6,5), (4,7), (7,4), (6,7), (7,6).}

Se $\forall v \in V(G)$, temos que $vv \notin E(G)$ dizemos que G é *simples*. todos os grafos tratados neste trabalho são simples e não orientados.

A *vizinhança* de um vértice $v \in V(G)$ em G é o conjunto dos vértices adjacentes a v, ou seja $N_G(v) = \{u \in V(G) \mid uv \in E(G)\}$. Inversamente, a *não vizinhança* de v é o conjunto $\bar{N}_G(v) = \{u \in V(G) \mid uv \notin E(G)\}$. Podemos extrapolar as definições acima para um conjunto de vértices $S \subseteq V(G)$, tendo os significados indicados a seguir:

- $N_G(S) = \{u \in V(G) \mid u \notin S, \exists v \in S \text{ tal que } uv \in E(G)\}$.

- $\bar{N}_G(S) = \{u \in V(G) \mid u \notin S, \forall v \in S \text{ tal que } uv \notin E(G)\}$.

Quando o grafo, ao qual as notações acima se referem, estiver claro pelo contexto, adotamos as notações mais compactas $N(v)$, $N\bar{(}v)$, $N(S)$ e $N\bar{(}S)$ em detrimento a $N_G(v)$, $N_{\bar{G}}(v)$, $N_G(S)$ e $N_{\bar{G}}(S)$.

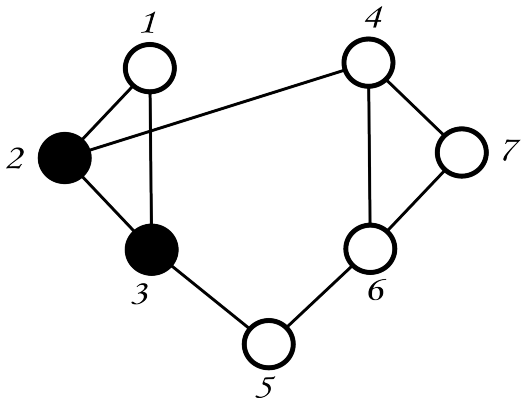


Figura 11: Vizinhança do vértice 1 em G_1, $N_{G_1}(1) = \{2, 3\}$.

Dado $S \subseteq V(G)$, o *grafo induzido por S em G* é o grafo $G[S] = (S, E[S])$ em que $E[S] = \{uv \in E \mid u, v \in S\}$. O *grau* de $v \in V$ em G é dado por $deg_G(v) = |N_G(v)|$. Se tomarmos $S \subseteq V$, então usamos a notação $deg_G(S)$ para representar $\sum_{v \in S} deg_G(v)$.

A.2 Conjunto Independente

Um conjunto de vértices $I \subseteq V(G)$ é um *conjunto independente ou estável* se seus elementos são mutuamente não adjacentes.

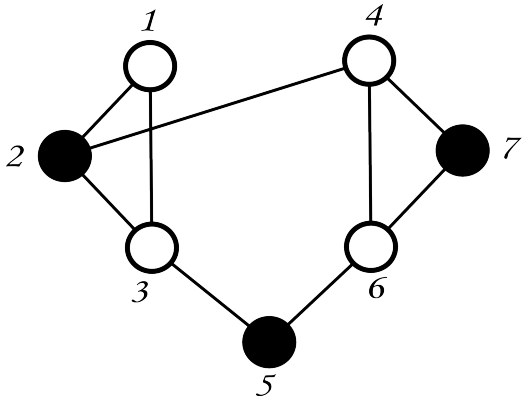


Figura 12: Conjunto Independente $\{2, 5, 7\}$ do grafo G_1.

Uma *clique* é um conjunto de vértices $C \subseteq V(G)$ tal que todos seus elementos são mutuamente adjacentes. O *complemento* de um grafo $G = (V, E)$ é o grafo $\overline{G} = (V(G), \overline{E(G)})$, em que $\overline{E(G)} = \{uv \mid u, v \in V(G); u \neq v; uv \notin E(G)\}$. Observe que, se $I \subseteq V(G)$ é um conjunto independente em G, este é uma clique de $\overline{G}$.

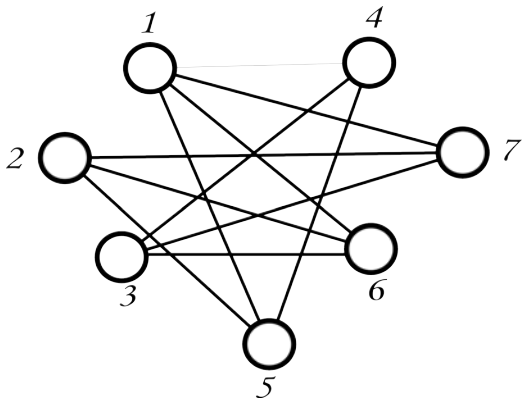


Figura 13: Complemento do grafo da Figura 11.

O conjunto independente I é *maximal* se não é subconjunto próprio de nenhum outro conjunto independente é *máximo* se, para todo conjunto independente X de G temos que $|I| \geq |X|$. A cardinalidade de um conjunto estável máximo de G é o *índice de estabilidade* $\alpha(G)$.

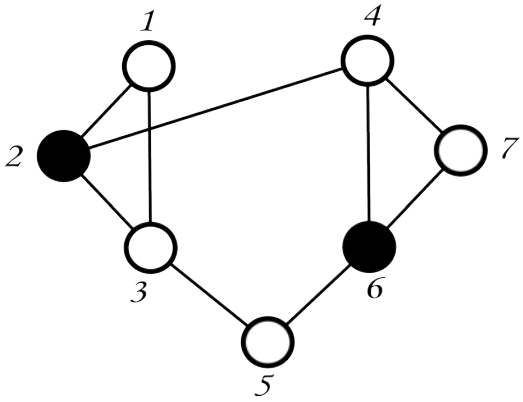


Figura 14: Conjunto independente $I = \{2, 6\}$ maximal. Observe que este conjunto não é máximo.

Convém agora, fazermos algumas observações sobre os conjuntos independentes maximais de G.

Observação 2. *Um conjunto independente $I \subseteq V(G)$ de G é maximal se, e somente se, $(V(G) \setminus I) \subseteq N(I)$.*

Observação 3. *Seja e I um conjunto independente maximal de G. Se existe $v \in V \setminus I$ tal que $N(v) \cap I = \{u\}$, então $(I \setminus \{u\}) \cup \{v\}$ é um conjunto independente, não necessariamente maximal, de G.*

A Observação 2 é imediata da definição de conjunto independente maximal e, para verificar a validade da Observação 3, basta observar que podemos trocar u por v no conjunto independente I.

A.3 Coloração de Vértices

Uma *k-coloração*, $k \geq 1$, de G é uma função $c : V(G) \to \{1, \ldots, k\}$ tal que vértices adjacentes recebem valores distintos, ou seja, se $uv \in E(G)$ então $c(v) \neq c(u)$. Usualmente o valor $c(v)$ é denominado de *cor* de v segundo c. Ao menor valor k para o qual G admite uma k-coloração, denominamos *número cromático* de G e o denotamos por $\chi(G)$.

Uma k-coloração de um grafo $G = (V, E)$ induz uma partição de $V(G)$ em k conjuntos independentes $\{I_1^c, \ldots, I_k^c\}$, onde

$$I_i^c = \{v \mid v \in V(G); c(v) = i\}.$$

Uma *k-coloração gulosa* g de G é uma coloração tal que se $v \in V(G)$ tem a cor i, então para todo $j < i$ existe um vizinho de v com a cor j, ou seja, $N(v) \cap I_j^g \neq \emptyset$. Uma maneira simples de construir uma coloração gulosa consiste em visitar os vértices de $V(G)$, em ordem crescente. Ao visitar um vértice tenta-se atribuir a este a menor cor disponível, ou seja, que não aparece em sua vizinhança, das cores já utilizadas. Se não existe cor disponível, criamos uma nova cor e atribuímos a v.

Algoritmo 14: Algoritmo Guloso por Vértices

Entrada: grafo $G = ([n], E)$.
Saída: coloração gulosa g de G

```
1  coloração g
2  para todo v ∈ [n] faça
3      g[v] ← 0
4  cor ← 1
5  C_cor ← ∅
6  para todo v ∈ [n] faça
7      ℓ ← cor + 1
8      C_ℓ ← ∅
9      para j de 1 até cor faça
10         se ∃j, 1 ≤ j ≤ cor tal que N(v) ∩ C_ℓ = ∅ então
11             g[v] ← ℓ
12             C_ℓ ← C_ℓ ∪ {v}
13             cor ← max{cor, ℓ}
14             ℓ ← j
15             j ← cor + 2
```

O Algoritmo 14 é uma *heurística gulosa por vértices*, no sentido que visita os vértices do grafo de entrada na ordem estabelecida. para respeitar a ordem de visitação dos vértices, todas as cores precisam ser guardadas (nos conjuntos $C_1, \ldots, C_{cor}$). Outra maneira de se obter uma coloração gulosa é uma *heurística gulosa por cores* que é apresentada no Algoritmo 15. neste algoritmo, as cores são determinadas uma após a outra. Por essa razão, somente o conjunto correspondente à cor corrente (C) precisa ser mantido.

Algoritmo 15: Algoritmo Guloso por Cores

```
1  Guloso(G, ≺_G)
   Entrada: grafo G = ([n], E)
   Saída: coloração gulosa g de G
2  coloração g
3  para todo v ∈ [n] faça
4      g[v)] ← 0
5  cor ← 1
6  S ← [n]
7  enquanto R ≠ ∅ faça
8      C ← ∅
9      Γ ← S
10     enquanto Γ ≠ ∅ faça
11         v ← menor vértice de Γ
12         g[v] ← cor
13         C ← C ∪ {v}
14         Γ ← Γ \ N(v)
15     S ← S \ C
16     cor ← cor + 1
17 retorna g
```

É possível verificar, através de uma argumento indutivo, que ambos os Algoritmos 14 e 15 obtêm a mesma coloração. Sejam cor_1 e cor_2 as colorações obtidas com os Algoritmos 14 e 15, respectivamente. Para mostrar que $cor_1 = cor_2$, suponha $v \in [n]$ tal que $cor_1[u] = cor_2[u]$ para todo $u < v$ (note que $cor_1[1] = cor_2[1]$). Pelo critério guloso adotado no Algoritmo 14 concluímos dois fatos. Primeiro, $u < v$ implica que $cor_1[u] \leq cor_1[v]$. Segundo, para todo $j < cor_1[v]$, existe $u \in N(v), u < v$ tal que $cor_1[u] = j$. Este segundo fato implica que $v \in R$, tomando-se o estado de R no início da iteração correspondente a $cor = cor_1[v]$. Já do primeiro fato e da hipótese de indução decorre que se $u < v$ é tal que $u \in R$, então $cor_1[u] = cor_i[v]$.

Finalmente, cabe o comentário que uma coloração dos vértices de G corresponde a uma *cobertura por cliques* (isto é, uma partição de V em cliques) de $\bar{G}$. Portanto, uma simples adaptação das heurísticas de coloração gulosa pode transformá-las em heurísticas gulosas para cobertura por cliques.