UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

BACHARELADO EM ENGENHARIA DE SOFTWARE

MAURO ROBERTO COSTA DA SILVA

ALGORITMOS PARA O PROBLEMA DO *K*-PLEX MÁXIMO UTILIZANDO PARALELISMO DE BITS E COLORAÇÃO

QUIXADÁ – CEARÁ

2016

MAURO ROBERTO COSTA DA SILVA

ALGORITMOS PARA O PROBLEMA DO *K*-PLEX MÁXIMO UTILIZANDO PARALELISMO DE BITS E COLORAÇÃO

Monografia apresentada no curso de Engenharia de Software da Universidade Federal do Ceará, como requisito parcial à obtenção do título de bacharel em Engenharia de Software. Área de concentração: Computação.

Orientador: Wladimir Araújo Tavares

QUIXADÁ – CEARÁ

2016

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S581a Silva, Mauro Roberto Costa da.
Algoritmos para o problema do k-plex máximo utilizando paralelismo de bits e coloração / Mauro Roberto Costa da Silva. – 2016.
42 f. : il.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2016.
Orientação: Prof. Dr. Wladimir Araújo Tavares.

1. Teoria dos Grafos. 2. Clique Problem. 3. Otimização Combinatória. I. Título.

CDD 005.1

MAURO ROBERTO COSTA DA SILVA

ALGORITMOS PARA O PROBLEMA DO K-PLEX MÁXIMO UTILIZANDO PARALELISMO DE BITS E COLORAÇÃO

Monografia apresentada no curso de Engenharia de Software da Universidade Federal do Ceará, como requisito parcial à obtenção do título de bacharel em Engenharia de Software. Área de concentração: Computação.

Aprovada em:

BANCA EXAMINADORA

Wladimir Araújo Tavares (Orientador)
Campus Quixadá
Universidade Federal do Ceará – UFC

Fábio Carlos Sousa Dias
Campus Quixadá
Universidade Federal do Ceará - UFC

Paulo Henrique Macêdo de Araújo
Campus Quixadá
Universidade Federal do Ceará - UFC

Aos meus pais.

AGRADECIMENTOS

Aos Meus Pais, Maria José e Jusmaro Correia.

Á toda minha família e amigos.

Ao Prof. Dr. Wladimir Araújo Tavares, pela excelente orientação, disponibilidade e paciência.

Aos professores participantes da banca examinadora Fábio Carlos Sousa Dias e Paulo Henrique Macêdo de Araújo pelo tempo, pelas valiosas colaborações e sugestões.

“Tenha talento, trabalhe como um condenado,
sue sangue, e você conseguirá tudo sem esforço.”
(Humberto Gessinger)

RESUMO

O problema do k-plex máximo é uma relaxação do problema da clique máxima (TRUKHANOV et al., 2013) e pode ser apresentado da seguinte maneira: dado um grafo não-direcionado $G = (V, E)$, encontre $S \subseteq V$ com cardinalidade máxima tal que $\forall v \in S, |\Gamma(v) \cap S| \geq |S| - k$, em que $\Gamma(v)$ representa o conjunto de vértices adjacentes de v. Algoritmos para encontrar o k-plex máximo em um grafo são usados, por exemplo, em redes sociais para analisar subgrupos coesos. A coesão social é usada para explicar e desenvolver teorias sociológicas (BALASUNDARAM; BUTENKO; HICKS, 2011). Neste trabalho, modificamos o algoritmo proposto por McClosky e Hicks (2012), chamado de *BasicPlex*, adicionando um procedimento de limite superior e uma estratégia de ramificação baseada em uma coloração obtida heuristicamente, a utilização de uma estrutura de dados chamada de vetores de bits e a utilização de listas de vértices saturadas para a geração do conjuntos candidatos. Comparamos empiricamente o limite superior apresentado neste trabalho com *IWCCH* de McClosky e Hicks (2012). Testes computacionais foram realizados. Apresentamos o algoritmo chamado *BasicPlexBranching* (*BPB*) que combina as técnicas listadas acima. Esse algoritmo foi comparado à *RDPlex*, solução proposta por Trukhanov et al. (2013). Foram utilizadas instâncias do DIMACS para realização dos testes e *BPB* obteve resultado melhor em 26 instâncias, enquanto *RDPlex* se saiu melhor em 21 instâncias.

Palavras-chave: Teoria dos Grafos. Clique Problem. Otimização Combinatória.

ABSTRACT

The maximum k-plex problem is a relaxation of the maximum clique problem (TRUKHANOV et al., 2013) and can be presented as follows: given an undirected graph $G = (V, E)$, find $S \subseteq V$ with a maximum cardinality such that $\forall v \in S, |\Gamma(v) \cap S| \geq |S| - k$, where $\Gamma(v)$ represents the set of adjacent vertices of v. Algorithms to find the maximum k-plex in a graph are used, for example, in social networks to analyze cohesive subgroups. Social cohesion is used to explain and develop sociological theories Balasundaram, Butenko e Hicks (2011). In this work, we modified the algorithm proposed by McClosky e Hicks (2012), called *BasicPlex*, by adding an upper bound procedure and a branching strategy based on a heuristically obtained coloring, using a data structure called bit vectors and using saturated vertex lists for the generation of candidate sets. We empirically compared the upper bound presented in this paper with *IWCCH* of McClosky e Hicks (2012). Computational tests were performed. We introduce the algorithm called *BasicPlexBranching* (*BPB*) that combines the techniques listed above. This algorithm was compared to the *RDPlex*, solution proposed by Trukhanov et al. (2013). DIMACS instances were used to perform the tests and *BPB* obtained best result in 26 instances, whereas *RDPlex* did best in 21 instances.

Keywords: Graph Theory. Clique Problem. Combinatorial Optimization

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BPB	BasicPlexBranching
DIMACS	Center for Discrete Mathematics and Theoretical Computer Science
PCM	Problema da Clique Máxima
RDS	Russian Doll Search

LISTA DE SÍMBOLOS

$\omega_k(G)$	Maior k-plex do grafo G
$\Gamma(v)$	Conjunto de vértices adjacente a v
$degree_{[G]}(v)$	Grau do vértice v no grafo G
$\Delta(G)$	Maior Grau do grafo G

SUMÁRIO

1 INTRODUÇÃO

O Problema da Clique Máxima (PCM) pode ser apresentado da seguinte maneira: dado um grafo não-direcionado $G = (V,E)$, encontre $C \subseteq V$ com cardinalidade máxima tal que $\forall u,v \in C, \{u,v\} \in E$ (ou seja, C é uma clique). O PCM é um problema NP-difícil. Em termos práticos, o PCM possui um grande número de aplicações em áreas como: bioinformatica, processamento de imagens, design de circuitos quânticos e no projeto de sequências de DNA e RNA para a computação biomolecular (TOMITA; SEKI, 2003).

O problema k-plex máximo é uma relaxação do problema da clique máxima (TRUKHANOV et al., 2013) e pode ser apresentado da seguinte maneira: dado um grafo não-direcionado $G = (V,E)$, encontre $S \subseteq V$ com cardinalidade máxima tal que $\forall v \in S, |\Gamma(v) \cap S| \geq |S| - k$, em que $\Gamma(v)$ representa o conjunto de vértices adjacentes de v. É fácil ver que toda clique é um 1-plex.

Algoritmos para encontrar o k-plex máximo em um grafo são usados, por exemplo, em redes sociais para analisar subgrupos coesos. A coesão social é usada para explicar e desenvolver teorias sociológicas (BALASUNDARAM; BUTENKO; HICKS, 2011). Os principais algoritmos existentes para esse problema foram propostos em McClosky e Hicks (2012) e Trukhanov et al. (2013).

Como o problema da clique máxima, o problema do k-plex máximo representa também um grande desafio algorítmico. Os algoritmos com os melhores resultados computacionais são os algoritmos combinatórios que utilizam *branch-and-bound* e o método das bonecas russas.

O objetivo geral deste trabalho é propor uma solução para o problema do k-plex máximo de forma mais eficiente que algumas soluções encontradas na literatura. Com a execução deste trabalho espera-se alcançar três objetivos específicos: (i) Desenvolver um algoritmo para o problema do k-plex; (ii) Encontrar uma heurística de coloração para o problema k-plex máximo tentando manter um bom equilíbrio entre o poder de poda e o tempo gasto para calculá-lo; (iii) Comparar a solução proposta com as encontradas em Trukhanov et al. (2013) e McClosky e Hicks (2012).

Neste trabalho, foram adaptadas técnicas oriundas dos algoritmos de clique máxima para o problema k-plex máximo, utilizando paralelismo de bits e heurísticas de coloração, comparando o desempenho das soluções propostas com as encontradas em McClosky e Hicks (2012) e em Trukhanov et al. (2013).

Os testes de desempenho foram realizados com instâncias obtidas do *benchmark* do DIMACS (Center for Discrete Mathematics and Theoretical Computer Science).

2 TRABALHOS RELACIONADOS

McClosky e Hicks (2012) apresenta um algoritmo combinatorial para o problema da clique máxima e uma adaptação dessa solução para o problema do *k*-plex máximo. McClosky e Hicks (2012) utiliza heurísticas de co-coloração para encontrar o limite superior para o problema em grafos ponderados, ou seja, grafos em que os vértices possuem pesos. As soluções propostas por McClosky e Hicks (2012), tanto para o problema da clique máxima quanto para o *k*-plex máximo não fazem uso do paralelismo de bits.

Trukhanov et al. (2013) propõem um *framework* para resolução de vários problemas que são relaxações do problema da clique máxima. Em Trukhanov et al. (2013), o *framework* proposto foi utilizado para resolver os problema do *k*-plex máximo e da clique *s*-deficiente máxima. A solução de Trukhanov et al. (2013) consiste em um algoritmo *branch-and-bound* genérico que utiliza a técnica *Russian Doll Search* (RDS). O método das Bonecas Russas consiste em resolver, iterativamente, uma sequência de subproblemas de maneira aninhada. De tal maneira, cada subproblema pode aparecer como subproblema dos próximos. O valor ótimo de cada subproblema resolvido é armazenado para fins de poda. Neste algoritmo, são utilizadas informações extras de cardinalidade da não-vizinhança de cada vértice para acelerar o processo de criação do novo conjunto candidato -que é o conjunto que contém os elementos que podem entrar na solução atual-, também chamada de passo de verificação. Os critérios de poda utilizados pelo algoritmo ficam por conta do limite superior trivial (cardinalidade do conjunto candidato) e dos limites obtidos pelos subproblemas já resolvidos e armazenados.

Tomita e Seki (2003) apresenta uma solução para o problema da clique máxima utilizando coloração. A técnica de coloração utilizada em Tomita e Seki (2003) será apresentada na seção 4.3. A coloração apresentada por Tomita e Seki (2003) possui duas principais funções: definir os vértices que devem ser ramificados e definir a ordem em que os vértices são ramificados. No algoritmo de Tomita e Seki (2003), os vértices são ramificados seguindo a ordem da maior para a menor cor. Nosso trabalho, adapta a coloração proposta por Tomita e Seki (2003).

Em Segundo, Rodríguez-Losada e Jiménez (2011) foi utilizada a técnica de paralelismo de bits para o desenvolvimento de uma solução para o problema da clique máxima. Segundo, Rodríguez-Losada e Jiménez (2011) compara os resultados obtidos com os de algoritmos semelhantes que não utilizam paralelismo de bits. Podemos destacar dois aspectos importantes no trabalho de Segundo, Rodríguez-Losada e Jiménez (2011): a ordem fixa dos vértices e o paralelismo de bits. A ordem fixa dos vértices fornece um critério para a escolha dos

vértices durante a heurística de ordenação. Como a ordem é fixa, essa ordem pode ser retomada em cada subproblema sem a necessidade de reordenação. O paralelismo de bits consegue reduzir o número de instruções para realizar atividades recorrentes no algoritmo.

As soluções propostas neste trabalho utilizarão a técnica de paralelismo de bits, assim como apresentada em Segundo, Rodríguez-Losada e Jiménez (2011), juntamente com o algoritmo para a resolução do k-plex máximo e apresentados em McClosky e Hicks (2012), a técnica de coloração proposta em Tomita e Seki (2003) e e aproveita as informações das listas saturadas para a definição do conjunto candidato, assim como apresentado em Trukhanov et al. (2013). Neste trabalho, combinamos as técnicas apresentadas nos algoritmos anteriores: heurísticas de coloração, lista de vértices saturados, ordem fixa e paralelismo de bits para o desenvolvimento de um novo algoritmo.

3 FUNDAMENTAÇÃO TEÓRICA

3.1 Problema do *k*-plex máximo

Dado um grafo $G = (V, E)$, em que V é um conjunto finito de vértices de G e E é um conjunto finito de pares de vértices distintos (u, v) - chamados de arestas -, uma *clique* em G é um grafo $S \subseteq G$ tal que S é um grafo completo, ou seja, para qualquer par de vértices (u, v) tal que $u \in S$ e $v \in S$, existe uma aresta de u para v. Uma clique máxima em um grafo é aquela que possui o maior número de vértices (TOMITA; SEKI, 2003). O tamanho da maior *clique* de um grafo G é dado por $\omega(G)$. Encontrar um *clique* máxima é um problema *NP*-difícil (TOMITA; SEKI, 2003).

O problema do k-plex máximo é uma relaxação do problema da *clique* máxima (TRUKHANOV et al., 2013). Dado o conjunto $degree_{[G]}$, no qual $degree_{[G]}(v)$ é o grau de um vértice v em um grafo G, um k-plex é um conjunto $S \subseteq V(G)$ que satisfaz a seguinte condição: $\{v|degree_{[S]}(v) \geq |S| - k\}$, $\forall v \in S$, em que k é um valor inteiro positivo. O k-plex máximo é o que possui maior número de vértices. 1-plexes são cliques (MCCLOSKY; HICKS, 2012).

Este trabalho utiliza as definições de *clique* e k-plex apresentadas em (TOMITA; SEKI, 2003) e McClosky e Hicks (2012), respectivamente, para a resolução do problema de encontrar um k-plex máximo.

3.2 Paralelismo de Bits

O Paralelismo de Bits é uma técnica que consiste no uso de um vetor de bits (*bit field*, *bitmap*, *bit string*, etc) para representar conjuntos. Essa técnica processa blocos de informações porque utiliza operações a nível de bits, e também reduz o espaço de representação na memória (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011).

Dado um conjunto C e uma palavra de N bits, a representação de C a nível de bits é um vetor BC com N bits, tal que, para cada bit b de BC, se $b \in C$, $BC[b] \leftarrow 1$, caso contrário $BC[b] \leftarrow 0$ (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011).

Considere uma palavra de 8 bits, e os conjuntos $A = \{1, 3, 5\}$ e $B = \{0, 1, 2, 4, 5, 7\}$. As representações a nível de bit para os conjuntos A e B são, respectivamente, $BA = 01010100$ e $BB = 11101101$. Operações de conjuntos, como união ($BA \cup BB$), interseção ($BA \cap BB$) e diferença ($BA - BB$), por exemplo, podem ser facilmente realizadas com operações bit a bit nos

conjuntos *BA* e *BB*:

A operação de união utiliza a disjunção lógica (operador *ou*):

$$BA \cup BB = BA|BB = \begin{array}{lll} BA & 01010100 & \{1, 3, 5\} \\ BB & 11101101 & \{0, 1, 2, 4, 5, 7\} \\ \hline & 11111101 & \{0, 1, 2, 3, 4, 5, 7\} \end{array}$$

A operação de interseção utiliza a conjunção lógica (operador *e*):

$$BA \cap BB = BA\&BB = \begin{array}{lll} BA & 01010100 & \{1, 3, 5\} \\ BB & 11101101 & \{0, 1, 2, 4, 5, 7\} \\ \hline & 01000100 & \{1, 5\} \end{array}$$

Para realizar a diferença, utiliza-se a conjunção lógica (operador *e*) e a negação lógica (operador *não*):

$$BA - BB = BA\&(\neg BB) = \begin{array}{lll} BA & 01010100 & \{1, 3, 5\} \\ \neg BB & 00010010 & \{3, 6\} \\ \hline & 00100000 & \{3\} \end{array}$$

Uma desvantagem da utilização do paralelismo de bits é que a operação de listar todos os elementos de um vetor de bits pode ser demorada. (SEGUNDO; RODRÍGUEZ-LOSADA; JIMÉNEZ, 2011).

Os vetores de bits, como apresentada em Segundo, Rodríguez-Losada e Jiménez (2011), foi utilizada neste trabalho para a representação de conjuntos de vértices na resolução do problema do *k*-plex máximo. Essa escolha deve-se ao fato de que operações lógicas, como as exemplificadas anteriormente, são capazes de processar blocos de informações, e essas operações serão mais frequentes que a necessidade de percorrer os vetores de bits, fazendo com que haja um ganho de desempenho.

3.3 Coloração em Grafos

A coloração em grafos consiste em colorir elementos de um grafo de acordo com determinada restrição. Neste trabalho essa técnica foi aplicada com o objetivo de definir um limite superior para os subproblemas e, a partir desse limite, podar a árvore de buscas a fim de reduzir o número de ramificações desnecessárias.

Dado um grafo $G = (V, E)$, um conjunto $C \subseteq V$ é dito independente se, para qualquer par de vértices (u, v), tal que $u \in C$ e $v \in C$, não existe um aresta de u para v em G (BERMAN; FUJITO, 1995).

A técnica de coloração proposta em Tomita e Seki (2003) consiste na criação de conjuntos independentes (chamado de cores) e na ordenação dos vértices de acordo com o número do conjunto em que foram inseridos.

A coloração utilizada em Tomita e Seki (2003) consiste no seguinte procedimento: para cada vértice $v \in G$, adicione-o ao conjunto independente C_0 se $C_0 \cup \{v\}$ também for um conjunto independente, caso contrário, a verificação deve ser realizada para o conjunto C_1, e assim por diante, até o conjunto C_K, em que K é o número de *cores* atual. Caso nenhum dos conjuntos aceitem que o vértice v seja adicionado, um novo conjunto é criado, $C_{K+1} = \{v\}$, e K é incrementado. O limite superior para cada conjunto é indicado pelo número da cor, ou seja, o conjunto C_J tem limite superior J, pois os vértices adicionados nesse conjunto possuem, no mínimo, J adjacentes.

Podemos obter um limite superior para o k-plex máximo utilizando uma coloração da seguinte maneira:

Dado $\Delta(G)$ como o grau máximo de um determinado grafo G e $\omega_k(G)$ como o tamanho do maior k-plex de G:

Lema 12. Todo grafo G satisfaz $\omega_k(G) \leq \Delta(G) + k$ (MCCLOSKY; HICKS, 2012).

Demonstração. Suponha que existe um k-plex $C \subseteq G$ tal que $|C| > \Delta(G) + k$. Escolha um vértice $v \in K$. Observe que $degree_{[C]}(v) \geq |C| - k$ pela definição de k-plex. Portanto, $degree_{[C]}(v) \geq |C| - k > \Delta(G)$, é uma contradição desde que $C \subseteq G$ (MCCLOSKY; HICKS, 2012). □

Todos os conjuntos C_J gerados pela coloração de Tomita e Seki (2003) são independentes, portanto $\Delta(C_J) = 0$. A partir do Lema 12:

$$\omega_k(C_J) \leq \Delta(C_J) + k$$

simplificando:

$$\omega_k(C_J) \leq k$$

Portanto, k também é um limite superior para C_J. Esse limite foi utilizado pela função *Branching* apresentada na seção 4.4.1.

Uma abordagem chamada de coloração co-k-plex ou co-coloração, foi apresentada em McClosky e Hicks (2012) e utiliza a definição de co-k-plexes. Um co-k-plex $\overline{R}$ é um grafo complementar de R, tal que R é um k-plex (MCCLOSKY; HICKS, 2012). Um grafo $\overline{R} = (V, \overline{E})$ é dito complementar de R = *(V, E)* se os dois grafos possuírem os mesmos vértices, e para qualquer par de vértices u e v, tal que $u \in V$ e $v \in V$, *(u, v)* $\in R$ se, e somente se *(u, v)* $\notin \overline{R}$ (BONDY; MURTY, 1976).

Na co-coloração, um conjunto candidato U é recebido como entrada e divido em cores, assim como na coloração proposta em Tomita e Seki (2003). O procedimento da co-coloração é semelhante ao da coloração apresentado anteriormente, porém as cores são co-k-plexes, ou seja, um vértice $v \in U$ será adicionado em uma cor C_k apenas se $C_k \cup \{v\}$ for um co-k-plex. McClosky e Hicks (2012) também propôs limites superiores para co-k-plexes e os utilizou para calcular os limites das cores. O limite do conjunto candidato U é dado pela soma dos limites das cores formadas a partir de G. Os detalhes da co-coloração serão apresentada na seção 4.4.2.

Técnicas de coloração foram utilizadas neste trabalho para tentar reduzir o número de ramificações desnecessárias. A seção 4.4 apresenta o emprego de coloração na solução aqui propostas.

4 ALGORITMO PROPOSTO

Os passos para o desenvolvimento e teste da solução proposta neste trabalho serão apresentadas nesta seção.

4.1 Implementação do algoritmo base

Um subproblema do Problema do k-Plex Máximo pode ser definido pela tupla (S, U, S_{max}), onde S é o k-Plex que está sendo construído, U é chamado de conjunto candidato, pois é formado pelos vértices que podem entrar no k-Plex em construção e S_{max} é a solução de maior cardinalidade encontrada.

Um subproblema é podado se $|S| + |U| \leq S_{max}$, ou seja, o conjunto candidato atual não consegue produzir um k-Plex que superer a melhor solução encontrada. Se um subproblema não for podado, um vértice v de U deve ser escolhido para compor S e o conjunto candidato deve ser atualizado para enumerar todos os k-Plexes contendo o vértice v.

Em McClosky e Hicks (2012) um algoritmo básico para a resolução do problema do k-plex máximo foi apresentado. Esse algoritmo foi utilizado como base para a criação da função *BasicPlex*, que é o procedimento principal do algoritmo desenvolvido neste trabalho. Na Figura 1 é apresentado o pseudocódigo dessa função.

Figura 1 – Algoritmo *Branch-and-Bound* para encontrar k-plexes

```
1  S ← ∅
   Function BasicPlex (S, U, k, S_max)
2      if |S| > |S_max| then
3          S_max ← S
       end
4      while U ≠ ∅ do
5          if |U| + |S| ≤ |S_max| then
6              return
           end
7          U ← U − {v} para algum v ∈ U
8          S ← S ∪ {v}
9          U' ← {u ∈ U : S ∪ {u} é um k-Plex }                    // O(|U| × |S|^2)
10         BasicPlex(S, U', k, S_max)
11         S ← S − {v}
       end
```

Fonte: McClosky e Hicks (2012) (com adaptações)

O procedimento *BasicPlex* é uma abordagem *branch-and-bound*, ou seja, uma abordagem que utiliza os limites superiores dos subproblemas com o objetivo de eliminar ramificações infrutíferas. Na implementação apresentada na Figura 1 o limite superior utilizado é $|U| + |S|$. Esse limite pode ser melhorado usando coloração. Uma técnica de coloração será adiciona ao *BasicPlex* na seção 4.4.

BasicPlex recebe um conjunto U que contém os vértices candidatos a entrar na solução S; um inteiro k, que indica qual o tipo do k-plex, ou seja, se $k = 2$, o algoritmo irá procurar *2*-plexes; um conjunto S_{max} que guarda a melhor solução encontrada até o momento.

O conjunto S, inicialmente vazio, é o conjunto utilizado para construir novas soluções. Quando $|S| > |S_{max}|$, uma solução melhor foi encontrada e S_{max} é atualizado para S. Para cada vértice v em U, v é adicionado à solução S e *BasicPlex* é chamado recursivamente para o conjunto candidato *U'*, que contém apenas vértices que podem entrar na nova solução $S \cup \{v\}$.

4.2 Ordem inicial dos vértices

Antes da execução dos algoritmos, os vértices são ordenados da forma *smallest-last*, que adiciona os vértices com menor grau ao final. Isso permite que ramos sejam cortados de forma mais rápida e que mais ramos possam ser podados (TRUKHANOV et al., 2013).

4.3 Geração de conjuntos candidatos

Quando um novo vértice é adicionado à solução atual, é necessário gerar um novo conjunto candidato. Em *BasicPlex*, essa geração de conjuntos candidato é feita na Linha 9 e possui um custo elevado, pois para cada vértice $u \in U$ é necessário adicioná-lo à solução S e verificar se forma um k-Plex. A complexidade dessa operação é da ordem de $O(|U| \times |S|^2)$. Essa operação teve sua complexidade reduzida utilizando uma técnica apresentada em Trukhanov et al. (2013).

Em Trukhanov et al. (2013) um algoritmo chamado *MakeSaturatedList* foi utilizado para gerar conjuntos candidatos. Neste trabalho uma função chamada *Generation* utiliza o algoritmo de Trukhanov et al. (2013) na geração de conjuntos candidatos. O pseudocódigo de *MakeSaturatedList* e *Generation* serão apresentados a seguir.

Figura 2 – Algoritmo para gerar lista de vértices saturados

```
    Function MakeSaturatedList(U, nncnt, S, k)
1   |  SL ← ∅
2   |  u ← último vértice de S
3   |  for each v ∈ S − {u} do
4   |  |  if (u,v) ∉ G then
5   |  |  |  nncnt[v]++
    |  |  end
    |  end
6   |  for each v ∈ U do
7   |  |  if (u,v) ∉ G then
8   |  |  |  nncnt[v]++
    |  |  end
    |  end
9   |  for each v ∈ S do
10  |  |  if nncnt[v] = k − 1 then
11  |  |  |  SL ← SL ∪ {v}
    |  |  end
    |  end
12  |  return SL
```

Fonte: Trukhanov et al. (2013) (com adaptações)

Dada uma solução $S \subseteq G$, um vértice $v \in S$ é chamado de saturado quando possui $k-1$ não-vizinhos em S. Uma lista saturada $SL \subseteq S$ é um conjunto formado pelos vértices saturados da solução S. A definição de lista saturada permite inferir que se adicionarmos um vértice u que não é vizinho de um vértice saturado $v \in SL$, $S \cup \{u\}$ deixa de ser um k-plex (TRUKHANOV et al., 2013).

Trukhanov et al. (2013) utilizou a definição de listas saturadas (*Saturated Lists*) apresentada anteriormente para desenvolver o procedimento *MakeSaturatedList*. Essa função recebe um conjunto de vértices U; um vetor *nncnt*, no qual $nncnt[v]$ é o número de não-vizinhos do vértice v; a solução atual S; e um inteiro k indicando o tipo do k-plex. E retorna a lista de vértices saturados em S.

O algoritmo *MakeSaturatedList* está descrito no pseudocódigo da Figura 2 e seu procedimento consiste em: atribuir a u o último vértice adicionado à solução S; percorrer todos os vértices de S que não são adjacentes a u, atualizando seu número de não-vizinhos; o número de não-vizinhos dos vértices pertencentes ao conjunto candidato U também são atualizados. Isso é importante para evitar que vértices que estão no conjunto candidato e que possuam k não-vizinhos não sejam adicionados à solução, pois não gerariam um k-plex; por fim, o algoritmo adiciona

todos os vértices pertencentes à S que são saturados, ou seja, que possuem $k-1$ não-vizinhos, à lista SL. A complexidade de *MakeSaturatedList* é da ordem de $O(|S|+|U|)$.

As Linhas 6-8 de *MakeSaturatedList* não existem no algoritmo de Trukhanov et al. (2013), e isso faz com que vértices que estão no conjunto candidato e que possuem menos de $|S|-k$ adjacentes sejam considerados válidos para a solução. A correção para esse problema foi apresentada em Gschwind et al. (2015).

Sendo SL a lista dos vértices saturados em uma solução S, é possível concluir que os únicos vértices pertencentes ao conjunto candidato U que podem entrar na solução são os que possuem adjacência com todos os vértices saturados de SL. A partir disso, pode-se verificar quais vértices de U fazem parte do novo conjunto candidato. A função para essa verificação está descrita na Figura 3.

Figura 3 – Algoritmo para verificar se um conjunto é k-plex

```
   Function IsPlex(SL, v)
1  |  for each u ∈ SL do
2  |  |  if (u,v) ∉ G then
3  |  |  |  return false
   |  |  end
   |  end
4  |  return true
```

Fonte: Trukhanov et al. (2013) (com adaptações)

IsPlex também foi definida em Trukhanov et al. (2013) e é um procedimento utilizado para verificar se um vértice de um conjunto candidato possui adjacência com todos os vértices de uma lista saturada. Caso contrário, esse vértice não é candidato. A complexidade de *IsPlex* é da ordem de $O(|SL|)$.

Figura 4 – Função para gerar conjuntos candidatos

```
   Function Generation (U, k, S, nncnt)
1  |  R ← ∅
2  |  SL ← MakeSaturatedList(U, nncnt, S, k)
3  |  for each v ∈ U do
4  |  |  if nncnt[v] > k − 1 then
5  |  |  |  continue
   |  |  end
6  |  |  if IsPlex(SL, v) then
7  |  |  |  R ← R ∪ {v}
   |  |  end
   |  end
8  |  return (R, nncnt)
```

Fonte: Elaborado pelo Autor

A função *Generation* recebe um conjunto candidato U e, para cada vértice $v \in U$, verifica se $nncnt[v] > k-1$, pois isso significaria que v não pode entrar na solução atual. Se $nncnt[v] <= K-1$, então a verificação *IsPlex* é executada para o vértice v. *Generation* retorna um conjunto R com todos os vértices pertencentes a U que satisfizeram as duas condições e o vetor *nncnt* atualizado. A complexidade de *Generation* é da ordem de $O(|U| \times |SL| + |S| + |U|)$.

BasicPlex foi alterado para utilizar a função *Generation* na geração de conjuntos candidatos. Essa alteração é mostrada na Figura 5.

Figura 5 – *BasicPlex* utilizando *Generation*

```
1  S ← ∅
   Function BasicPlex (S, U, k, S_max, nncnt)
2  |  if |S| > |S_max| then
3  |  |  S_max ← S
   |  end
4  |  while U ≠ ∅ do
5  |  |  if |U| + |S| ≤ |S_max| then
6  |  |  |  return
   |  |  end
7  |  |  v ← primeiro vértice de U
8  |  |  S ← S ∪ {v}
9  |  |  U ← U − {v}
10 |  |  (U', newNncnt) ← Generation(U, k)
11 |  |  BasicPlex(S, U', k, S_max, newNncnt)
12 |  |  S ← S − {v}
   |  end
```

Fonte: McClosky e Hicks (2012) (com adaptações)

4.4 Adicionando coloração

Neste trabalho a coloração foi utilizada para calcular o limite superior dos subproblemas e podar ramificações infrutíferas. Foram considerados dois algoritmos: um algoritmo chamado de *Branching* que utiliza uma técnica de coloração baseada em Tomita e Seki (2003) e o algoritmo *IWCCH* que foi apresentado em McClosky e Hicks (2012). Um teste de desempenho foi realizado entre os dois algoritmos, os resultados estão na Tabela 1.

Na maioria dos testes, *Branching* não resulta em um limite superior melhor que o calculado por *IWCCH*, porém o custo de *Branching* é inferior, o que é muito importante, tendo em vista que o cálculo será realizado para cada subproblema. Foi possível observar também que *Branching* obtém um limite melhor que *IWCCH* em alguns grafos, como é o caso do grafo *san1000*. Portanto, *Branching* foi a técnica utilizada na solução proposta neste trabalho.

Branching e *IWCCH* serão apresentados, respectivamente, nas seções 4.4.1 e 4.4.2.

4.4.1 Branching

A função *Branching* é baseada na técnica de coloração de Tomita e Seki (2003), apresentada na seção 4.3 e no Lema 12. *Branching*, mostrada na Figura 8, recebe um conjunto de vértices U e um inteiro *limite*, que representa o limite superior mínimo para que o conjunto U gere uma solução melhor que S_{max}. Um conjunto independente $C \subseteq G$ é dito maximal quando $\{\nexists\, v \,|v \notin C, v \in U, C \cup \{v\}$ é independente$\}$.

Figura 6 – Algoritmo *Branching*

```
    Function Branching (U, limite)
1   |  UB ← limite
2   |  j ← 0
3   |  while U ≠ ∅ do
4   |  |  if UB ≤ 0 then
5   |  |  |  return U
    |  |  end
6   |  |  S_j ← ∅
7   |  |  [R ← U]
8   |  |  for each v ∈ R do
9   |  |  |  S_j ← S_j ∪ {v}
10  |  |  |  [R ← R − Γ(v)]
11  |  |  |  [R ← R − {v}]
    |  |  end
12  |  |  count ← min(|S_j|, k, UB)
13  |  |  UB ← UB − count
14  |  |  U ← U − os primeiros count elementos de S_j
15  |  |  j++
    |  end
16  |  return U
```

Fonte: Tomita e Seki (2003) (com adaptações)

Branching gera conjuntos independentes maximais de U e remove os primeiros *limite* elementos de U utilizando a ordem dos conjuntos gerados. Ao final desse processo, retorna o conjunto U alterado. A complexidade da função *Branching* é da ordem de $O(|U|^2)$.

Na linha 10, $\Gamma(v)$ é o conjunto de vértices adjacentes a v. As linhas em caixas representam operações realizadas com paralelismo de bits.

Figura 7 – Algoritmo *BasicPlex* utilizando a função de Branching

```
1  S ← ∅
   Function BasicPlexBranching (S, U, k, S_max, nncnt)
2      if |S| > |S_max| then
3          S_max ← S
       end
4      limite ← |S_max| − |S|
5      R ← Branching(U, limite)
6      while R ≠ ∅ do
7          v ← primeiro vértice de R
8          S ← S ∪ {v}
9          U ← U − {v}
10         R ← R − {v}
11         (U', newNncnt) ← Generation(U, k, S, nncnt)
12         BasicPlexBranching(S, U', k, S_max, newNncnt)
13         S ← S − {v}
       end
```

Fonte: McClosky e Hicks (2012) (com adaptações)

A função *BasicPlexBranching* utiliza a função *Branching* para, a partir do conjunto candidato U, gerar um conjunto R que contém apenas os vértices que devem ser ramificados. O *limite* passado para *Branching* é a diferença entre o tamanho da melhor solução e o tamanho da solução atual ($|S_{max}| - |S|$). A chamada à função *Branching* irá retornar todos os vértices de U que possuem mais do que *limite* vizinhos, ou seja, que podem tornar a solução atual (S) melhor que a melhor solução encontrada até o momento (S_{max}).

4.4.2 IWCCH

Dado um co-k-plex $C \subseteq G$, em McClosky e Hicks (2012) foram definidos alguns limites superiores para C, são eles:

a) O tamanho de C, dado por $|C|$;

b) $2k - 2 + k \ mod \ 2$, em que k é um inteiro positivo que representa o tipo do co-k-plex;

c) $\Delta(G[C]) + k$, em que $\Delta(G[C])$ é maior grau de C, e k é um inteiro que representa o tipo do k-plex;

d) $k + J$, em que $J \leftarrow max\{m : a_m \geq k + m\}$ e a_m é o número de vértices com grau maior ou igual a m.

Esses limites foram apresentados e demonstrados em McClosky e Hicks (2012), e são a base para o funcionamento do algoritmo *IWCCH* apresentado na Figura 8.

Figura 8 – Algoritmo *IWCCH*

```
   Function IWCCH (U, limite)
1  | max ← 0
2  | for each v ∈ R do
3  | |  m ← min{i | IsCoKPlex(S_i ∪ {v}, k)}
4  | |  S_m ← S_m ∪ {v}
5  | |  if m > max then
6  | |  |  max ← m
   | |  end
   | end
7  | for i ← 0 to max do
8  | |  J_i ← max{m : a_m ≥ k + m} para S_i
   | end
9  | bound ← 0
10 | for i ← 0 to max do
11 | |  bound ← bound + min{2k − 2 + k mod 2, k + J_i, Δ(G[S_i]) + k, |S_i|}
   | end
12 | return bound
```

Fonte: McClosky e Hicks (2012) (com adaptações)

A função *IWCCH*, apresentada na Figura 8, recebe um conjunto de vértices candidatos U e um inteiro k que representa o tipo do k-plex. *IWCCH* forma conjuntos de vértices que são co-k-plex, ou seja, que seus complementos são k-plexes. Após formar todos os conjuntos, o algoritmo utiliza os limites superiores apresentados anteriormente para calcular o limite de cada co-k-plex, acumulando esse valor na variável *bound*, que ao final do processo será o limite do subproblema. Na linha 11 da função *IWCCH* o limite superior para cada subconjunto é dado pelo menor dos limites apresentados. A complexidade dessa função é da ordem de $O(|U|^2)$.

Na geração dos conjuntos, uma função chamada *IsCoKPlex* foi utilizada, o seu pseudocódigo será apresentado na Figura 9.

Figura 9 – Função para verificar se um conjunto é um co-k-plex

```
   Function IsCoKPlex (S, k)
1  |  for each v ∈ S do
2  |  |  if [degree_S[v]] ≥ k then
3  |  |  |  return false
   |  |  end
   |  end
4  |  return true
```

Fonte: McClosky e Hicks (2012) (com adaptações)

A função *IsCoKPlex* recebe um conjunto de vértices S e um inteiro k, que é o tipo do co-k-plex e verifica para cada v em S se o grau de v é menor que o valor de k, caso exista algum vértice que não satisfaça essa condição, S não é um co-k-plex. É possível utilizar paralelismo de bits para calcular o grau de cada vértice, o que possibilita que *IsCoKPlex* tenha complexidade de $O(|S|)$.

A Tabela 1 apresenta a comparação dos algoritmos de coloração *Branching* e *IWCCH*. É possível observar que *IWCCH* possui um limite superior mais aproximado na maioria das entradas, porém *Branching* apresenta um melhor resultado quando a comparação é realizada a partir do tempo de execução. Os testes foram realizados apenas para $k = 2$.

Tabela 1 – Resultados dos algoritmos de *Branching* e *IWCCH*

Grafo	Branching		IWCCH	
	$\omega_2(G)$	segundos	$\omega_2(G)$	segundos
brock200_1	107	0.00062	92	0.00087
brock200_2	66	0.00077	58	0.00102
brock200_3	82	0.00093	72	0.00098
brock200_4	94	0.00090	82	0.00096
brock400_1	193	0.00252	173	0.00369
brock400_2	192	0.00252	168	0.00313
brock400_3	194	0.00250	172	0.00344
brock400_4	194	0.00251	173	0.00346
brock800_1	281	0.00998	251	0.01471
brock800_2	284	0.00989	250	0.01456
brock800_3	276	0.01001	252	0.01447
brock800_4	281	0.00995	254	0.01459
C125.9	105	0.00025	89	0.00030
C250.9	194	0.00082	170	0.00118
C500.9	342	0.00309	296	0.00479

Continua na próxima página...

Grafo	Branching		IWCCH	
	$\omega_2(G)$	segundos	$\omega_2(G)$	segundos
C1000.9	621	0.01060	534	0.02111
C2000.5	439	0.06773	393	0.12060
C2000.9	1159	0.04134	1012	0.12238
C4000.5	794	0.27126	720	0.63469
c-fat200-1	27	0.00038	18	0.00097
c-fat200-2	46	0.00038	36	0.00093
c-fat200-5	142	0.00037	87	0.00091
c-fat500-1	28	0.00160	21	0.00534
c-fat500-2	52	0.00234	39	0.00517
c-fat500-5	128	0.00176	96	0.00407
c-fat500-10	252	0.00177	189	0.00416
gen200_p0.9_44	110	0.00050	146	0.00074
gen200_p0.9_55	134	0.00049	136	0.00071
gen400_p0.9_55	120	0.00207	240	0.00317
gen400_p0.9_65	138	0.00196	229	0.00296
gen400_p0.9_75	164	0.00193	233	0.00296
hamming6-2	64	0.00009	32	0.00011
hamming6-4	16	0.00010	8	0.00013
hamming8-2	256	0.00064	128	0.00112
hamming8-4	64	0.00116	32	0.00137
hamming10-2	1024	0.00647	512	0.01805
hamming10-4	256	0.00997	128	0.01362
johnson8-2-4	12	0.00003	12	0.00004
johnson8-4-4	41	0.00011	36	0.00013
johnson16-2-4	28	0.00022	56	0.00034
johnson32-2-4	60	0.00287	240	0.00409
keller4	62	0.00057	61	0.00060
keller5	221	0.00597	274	0.01039
MANN_a9	33	0.00005	40	0.00006
MANN_a27	261	0.00116	369	0.00316
MANN_a45	705	0.00652	1012	0.02899
MANN_a81	2241	0.06180	3279	0.69724
p_hat300-1	47	0.00141	41	0.00168
p_hat300-2	91	0.00164	76	0.00197
p_hat300-3	146	0.00146	129	0.00187
p_hat500-1	68	0.00378	62	0.00480
p_hat500-2	135	0.00441	120	0.00544
p_hat500-3	225	0.00382	198	0.00524
p_hat700-1	86	0.00716	78	0.00917
p_hat700-2	180	0.00850	158	0.00987

Continua na próxima página...

Grafo	Branching		IWCCH	
	$\omega_2(G)$	segundos	$\omega_2(G)$	segundos
p_hat700-3	298	0.00662	260	0.01027
p_hat1000-1	110	0.01302	99	0.01870
p_hat1000-2	234	0.01553	206	0.02081
p_hat1000-3	393	0.01347	344	0.02167
p_hat1500-1	154	0.02890	142	0.04624
p_hat1500-2	337	0.03417	304	0.05010
p_hat1500-3	565	0.02901	499	0.05489
san200_0.7_1	60	0.00069	105	0.00085
san200_0.7_2	36	0.00063	134	0.00089
san200_0.9_1	139	0.00052	135	0.00076
san200_0.9_2	120	0.00059	127	0.00087
san200_0.9_3	88	0.00062	133	0.00089
san400_0.5_1	26	0.00283	214	0.00377
san400_0.7_1	80	0.00250	200	0.00378
san400_0.7_2	60	0.00246	205	0.00376
san400_0.7_3	44	0.00251	216	0.00373
san400_0.9_1	202	0.00188	222	0.00290
san1000	30	0.01287	531	0.02607
sanr200_0.7	100	0.00072	84	0.00084
sanr200_0.9	156	0.00056	136	0.00078
sanr400_0.5	116	0.00318	102	0.00399
sanr400_0.7	178	0.00270	156	0.00358

4.5 Algoritmo utilizando Russian-Doll-Search

A técnica Russian Doll Search (RDS) é uma técnica que consiste em resolver um problema de tamanho N a partir da resolução de N sucessivos subproblemas. O primeiro subproblema é composto apenas do N-ésimo elemento, o segundo subproblema é composto dos dois últimos elementos, e assim sucessivamente até que sejam incluídos todos os N elementos e o N-ésimo subproblema seja igual ao problema original (VASKELAINEN et al., 2010).

Trukhanov et al. (2013) apresentou um algoritmo para o problema do k-plex máximo utilizando RDS. Esse algoritmo é o *RDPlex* apresentado na Figura 10.

Figura 10 – Algoritmo *Russian Doll Search* para encontrar k-plexes

```
1  S_max ← ∅
2  record ← []
   Function RDPlex (G, k)
3      ordena os vértices de G, colocando o menor grau no fim (smallest-last)
4      U ← ∅
5      for i ← |G| downto 1 do
6          S ← {v_i}
7          for j ← i to |G| do
8              nncnt_[j] ← 0
           end
9          (U', newNncnt) ← Generation(U, k, S, nncnt)
10         Expand(U', k, S, newNncnt)
11         record[v_i] ← |S|
12         U ← U ∪ {v_i}
       end
```

Fonte: Trukhanov et al. (2013) (com adaptações)

RDPlex guarda os resultados dos subproblemas já calculados em *record*. Esses resultados são utilizados pela função *Expand* para podar subproblemas. O funcionamento de *Expand* é semelhante ao de *BasicPlex*, porém possui uma poda adicional resultante do algoritmo RDS. A função *Expand* está descrita na Figura 11.

Figura 11 – Função *Expand* utilizada em *RDPlex*

```
    Function Expand (U, k, S, nncnt)
1   |  if U = ∅ then
2   |  |  if |S| > |S_max| then
3   |  |  |  S_max ← S
    |  |  end
    |  end
4   |  while U ≠ ∅ do
5   |  |  if |U| + |S| ≤ |S_max| then
6   |  |  |  return
    |  |  end
7   |  |  i ← min{j : v_j ∈ U}
8   |  |  if record[i] + |S| < |S_max| then
9   |  |  |  return
    |  |  end
10  |  |  U ← U − {v_i}
11  |  |  S ← S ∪ {v_i}
12  |  |  (U', newNncnt) ← Generation(U, k, S, nncnt)
13  |  |  Expand(U', k, S, newNncnt)
14  |  |  S ← S − {v_i}
    |  end
```

Fonte: Trukhanov et al. (2013) (com adaptações)

Em Trukhanov et al. (2013) os resultados computacionais de *RDPlex* foram comparados com os resultados de McClosky e Hicks (2012) e se mostrou superior. Portanto, neste trabalho a solução proposta será comparada somente ao algoritmo *RDPlex* de Trukhanov et al. (2013).

4.6 Realização dos Testes

Para a execução dos testes foram utilizadas entradas do *DIMACS* e instâncias geradas aleatoriamente. A descrição das entradas está no Apêndice A.

O algoritmo *RDPlex* de Trukhanov et al. (2013) foi implementado e executado no mesmo computador que a solução proposta neste trabalho para que comparações fossem realizadas de forma mais aproximada. Em Trukhanov et al. (2013) não foi utilizado paralelismo de bits, então o seu algoritmo foi implementado utilizando vetores de inteiros.

Os resultados estão apresentados na seção 5.

5 RESULTADOS COMPUTACIONAIS

Os testes foram realizados em um computador com 8GB de memória RAM, processador Intel(R) Xeon(R) CPU E31240 @ 3.30GHz e Sistema Operacional Linux.

A Tabela 2 apresenta a comparação dos algoritmos *BasicPlexBranching (BPB)* e *RDPLex*. Como *RDPlex* utiliza o método das bonecas russas (RDS) e *BPB* utiliza *Branch-and-bound*, comparações a respeito do número de subproblemas não fazem sentido.

Os testes foram realizados com tempo limite de 3600 segundos e apenas para $k = 2$. Os valores destacados indicam qual algoritmo obteve melhor resultado.

Tabela 2 – Resultados dos algoritmos de *BasicPlexBranching* e *RDPlex*

Grafo	BPB		RDPlex	
	$\omega_2(G)$	segundos	$\omega_2(G)$	segundos
brock200_1	26	>3600	26	>3600
brock200_2	13*	10.72	**13***	**7.13**
brock200_3	17*	211.35	**17***	**110.80**
brock200_4	20*	1115.30	**20***	**567.13**
brock400_1	**29**	**>3600**	28	>3600
brock400_2	28	>3600	28	>3600
brock400_3	**29**	**>3600**	28	>3600
brock400_4	**29**	**>3600**	28	>3600
brock800_1	**24**	**>3600**	23	>3600
brock800_2	**23**	**>3600**	22	>3600
brock800_3	**24**	**>3600**	23	>3600
brock800_4	**23**	**>3600**	22	>3600
C125.9	43	>3600	43	>3600
C250.9	**51**	**>3600**	48	>3600
C500.9	60	>3600	60	>3600
C1000.9	**71**	**>3600**	68	>3600
C2000.5	**18**	**>3600**	17	>3600
C2000.9	**79**	**>3600**	72	>3600
C4000.5	**19**	**>3600**	17	>3600
c-fat200-1	12*	0.02	**12***	**0.01**
c-fat200-2	24*	0.11	**24***	**0.01**
c-fat200-5	58	>3600	58	>3600
c-fat500-1	14*	0.26	**14***	**0.07**
c-fat500-2	26*	2.13	**26***	**0.08**
c-fat500-5	64	>3600	**64***	**54.28**
c-fat500-10	126	>3600	126	>3600

Continua na próxima página...

Grafo	BPB		RDPlex	
	$\omega_2(G)$	segundos	$\omega_2(G)$	segundos
gen200_p0.9_44	**49**	**>3600**	47	>3600
gen200_p0.9_55	52	>3600	**53**	**>3600**
gen400_p0.9_55	**59**	**>3600**	56	>3600
gen400_p0.9_65	64	>3600	64	>3600
gen400_p0.9_75	74	>3600	74	>3600
hamming6-2	32*	31.30	**32***	**2.57**
hamming6-4	6*	0.01	**6***	**0.00**
hamming8-2	128	>3600	128	>3600
hamming8-4	16	>3600	16	>3600
hamming10-2	512	>3600	512	>3600
hamming10-4	**44**	**>3600**	32	>3600
johnson8-2-4	5*	0.00	5*	0.00
johnson8-4-4	14*	3.56	**14***	**2.28**
johnson16-2-4	10	>3600	**10***	**2698.35**
johnson32-2-4	20	>3600	20	>3600
keller4	15*	448.00	**15***	**447.23**
keller5	**31**	**>3600**	30	>3600
MANN_a9	**26***	**0.21**	26*	0.70
MANN_a27	**235**	**>3600**	234	>3600
MANN_a45	**661**	**>3600**	660	>3600
p_hat300-1	10*	1.45	**10***	**0.49**
p_hat300-2	30	>3600	**30***	**412.73**
p_hat300-3	43	>3600	43	>3600
p_hat500-1	12*	21.49	**12***	**8.95**
p_hat500-2	42	>3600	42	>3600
p_hat500-3	**60**	**>3600**	59	>3600
p_hat700-1	13*	148.06	**13***	**51.60**
p_hat700-2	51	>3600	51	>3600
p_hat700-3	71	>3600	71	>3600
p_hat1000-1	13*	1133.85	**13***	**503.34**
p_hat1000-2	**55**	**>3600**	54	>3600
p_hat1000-3	**78**	**>3600**	77	>3600
p_hat1500-1	14	>3600	14	>3600
p_hat1500-2	**74**	**>3600**	71	>3600
p_hat1500-3	**111**	**>3600**	109	>3600
san200_0.7_1	30	>3600	30	>3600
san200_0.7_2	24	>3600	24	>3600
san200_0.9_1	90	>3600	90	>3600
san200_0.9_2	70	>3600	70	>3600
san200_0.9_3	48	>3600	48	>3600

Continua na próxima página...

Grafo	BPB		RDPlex	
	$\omega_2(G)$	segundos	$\omega_2(G)$	segundos
san400_0.5_1	14	>3600	14	>3600
san400_0.7_1	40	>3600	40	>3600
san400_0.7_2	30	>3600	30	>3600
san400_0.7_3	24	>3600	24	>3600
san400_0.9_1	100	>3600	100	>3600
san1000	16	>3600	16	>3600
sanr200_0.7	22	>3600	**22***	**2957.70**
sanr200_0.9	**50**	**>3600**	49	>3600
sanr400_0.5	15*	2210.96	**15***	**1380.06**
sanr400_0.7	**26**	**>3600**	25	>3600

*solução ótima

Para cada entrada foram destacadas as informações do algoritmo que obteve melhor resultado. Nos casos em que ambos algoritmos ultrapassaram o tempo limite de 3600 segundos, o algoritmo com a melhor solução viável encontrada foi considerado melhor. *BPB* obteve melhor resultado em 26 instâncias, enquanto *RDPlex* se saiu melhor em 21. Nas demais instancias os resultados de ambos algoritmos foram considerados equivalentes.

Em algumas entradas, como a *hamming10-4*, *BPB* encontrou um solução muito superior à encontrada por *RDPlex*. Existem algumas entradas, como a *p_hat300-2*, em que *RDPlex* conseguiu encontrar a solução ótima e *BPB* não. Apesar disso, *RDPlex* encontrou um $\omega_2(G)$ maior que *BPB* apenas na entrada *gen200_p0.9_55*.

Os resultados se mostraram satisfatórios, tendo em vista que a solução proposta neste trabalho apresentou desempenho melhor que a proposta em Trukhanov et al. (2013), consequentemente também apresentou melhor resultado computacional que a solução de McClosky e Hicks (2012).

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresenta um novo algoritmo para o problema do *k*-plex máximo, utilizando utilizando coloração e paralelismo de bits, que comparado ao estado da arte do problema, alcançou resultados melhores, do total de 76 instâncias, nosso algoritmo encontrou 26 soluções melhores e em 26 das demais instâncias, obteve resultado semelhante.

Uma pesquisa focada no limite superior poderia resultar em um algoritmo de coloração mais eficiente que *Branching*. Por questões de tempo, essa pesquisa não pôde ser realizada durante este trabalho, assim como algumas melhorias e passos. Estes serão apresentados a seguir como trabalhos futuros:

a) Desenvolver uma solução utilizando o método da boneca russa (RDS), paralelismo de bits e a coloração *Branching*;
b) Realizar comparação entre *RDPlex* e *BPB* com $k = 3$ e $k = 4$;
c) Encontrar um limite superior menor que *k* para cada conjunto independente de *Branching*.

Os trabalhos utilizados como referências foram de extrema importância, tendo em vista que a solução aqui apresentada utilizou dos conhecimentos descritos em tais trabalhos para ser desenvolvida.

REFERÊNCIAS

BALASUNDARAM, B.; BUTENKO, S.; HICKS, I. V. Clique relaxations in social network analysis: The maximum k-plex problem. **Operations Research**, INFORMS, v. 59, n. 1, p. 133–142, 2011.

BERMAN, P.; FUJITO, T. On approximation properties of the independent set problem for degree 3 graphs. In: **Algorithms and Data Structures**. Kingston, Canada: Springer, 1995. p. 449–460.

BONDY, J. A.; MURTY, U. S. R. **Graph theory with applications**. Londres: Macmillan London, 1976. v. 290.

GSCHWIND, T.; IRNICH, S.; PODLINSKI, I. et al. **Maximum weight relaxed cliques and Russian doll search revisited**. [S.l.], 2015.

MCCLOSKY, B.; HICKS, I. V. Combinatorial algorithms for the maximum k-plex problem. **Journal of combinatorial optimization**, Springer, v. 23, n. 1, p. 29–49, 2012.

SEGUNDO, P. S.; RODRÍGUEZ-LOSADA, D.; JIMÉNEZ, A. An exact bit-parallel algorithm for the maximum clique problem. **Computers & Operations Research**, Elsevier, v. 38, n. 2, p. 571–581, 2011.

TOMITA, E.; SEKI, T. An efficient branch-and-bound algorithm for finding a maximum clique. In: **Discrete mathematics and theoretical computer science**. Dijon, France: Springer, 2003. p. 278–289.

TRUKHANOV, S.; BALASUBRAMANIAM, C.; BALASUNDARAM, B.; BUTENKO, S. Algorithms for detecting optimal hereditary structures in graphs, with application to clique relaxations. **Computational Optimization and Applications**, Springer, v. 56, n. 1, p. 113–130, 2013.

VASKELAINEN, V. et al. Russian doll search algorithms for discrete optimization problems. Aalto-yliopiston teknillinen korkeakoulu, 2010.

APÊNDICE A – INSTÂNCIAS

Tabela 3 – Descrição das Instâncias utilizadas nos testes

Grafo	$\|V\|$	$\|E\|$	Densidade
brock200_1	200	14834	0.74543
brock200_2	200	9876	0.49628
brock200_3	200	12048	0.60543
brock200_4	200	13089	0.65774
brock400_1	400	59723	0.74841
brock400_2	400	59786	0.74920
brock400_3	400	59681	0.74788
brock400_4	400	59765	0.74893
brock800_1	800	207505	0.64926
brock800_2	800	208166	0.65133
brock800_3	800	207333	0.64873
brock800_4	800	207643	0.64970
C125.9	125	6963	0.89845
C250.9	250	27984	0.89908
C500.9	500	112332	0.90046
C1000.9	1000	450079	0.90106
C2000.5	2000	999836	0.50017
C2000.9	2000	1799532	0.90022
C4000.5	4000	4000268	0.50016
c-fat200-1	200	1534	0.07709
c-fat200-2	200	3235	0.16256
c-fat200-5	200	8473	0.42578
c-fat500-1	500	4459	0.03574
c-fat500-2	500	9139	0.07326
c-fat500-5	500	23191	0.18590
c-fat500-10	500	46627	0.37376
gen200_p0.9_44	200	17910	0.90000
gen200_p0.9_55	200	17910	0.90000
gen400_p0.9_55	400	71820	0.90000
gen400_p0.9_65	400	71820	0.90000
gen400_p0.9_75	400	71820	0.90000
hamming6-2	64	1824	0.90476
hamming6-4	64	704	0.34921
hamming8-2	256	31616	0.96863
hamming8-4	256	20864	0.63922
hamming10-2	1024	518656	0.99022
hamming10-4	1024	434176	0.82893

Continua na próxima página...

Grafo	$\|V\|$	$\|E\|$	Densidade
johnson8-2-4	28	210	0.55556
johnson8-4-4	70	1855	0.76812
johnson16-2-4	120	5460	0.76471
johnson32-2-4	496	107880	0.87879
keller4	171	9435	0.64912
keller5	776	225990	0.75155
MANN_a9	45	918	0.92727
MANN_a27	378	70551	0.99015
MANN_a45	1035	533115	0.99630
p_hat300-1	300	10933	0.24377
p_hat300-2	300	21928	0.48892
p_hat300-3	300	33390	0.74448
p_hat500-1	500	31569	0.25306
p_hat500-2	500	62946	0.50458
p_hat500-3	500	93800	0.75190
p_hat700-1	700	60999	0.24933
p_hat700-2	700	121728	0.49756
p_hat700-3	700	183010	0.74805
p_hat1000-1	1000	122253	0.24475
p_hat1000-2	1000	244799	0.49009
p_hat1000-3	1000	371746	0.74424
p_hat1500-1	1500	284923	0.25343
p_hat1500-2	1500	568960	0.50608
p_hat1500-3	1500	847244	0.75361
san200_0.7_1	200	13930	0.70000
san200_0.7_2	200	13930	0.70000
san200_0.9_1	200	17910	0.90000
san200_0.9_2	200	17910	0.90000
san200_0.9_3	200	17910	0.90000
san400_0.5_1	400	39900	0.50000
san400_0.7_1	400	55860	0.70000
san400_0.7_2	400	55860	0.70000
san400_0.7_3	400	55860	0.70000
san400_0.9_1	400	71820	0.90000
san1000	1000	250500	0.50150
sanr200_0.7	200	13868	0.69688
sanr200_0.9	200	17863	0.89764
sanr400_0.5	400	39984	0.50105
sanr400_0.7	400	55869	0.70011