UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ALAN MARTINS SILVA

UM ALGORITMO EXATO PARA O PROBLEMA DA BICLIQUE INDUZIDA BALANCEADA MÁXIMA

QUIXADÁ – CEARÁ

2016

ALAN MARTINS SILVA

UM ALGORITMO EXATO PARA O PROBLEMA DA BICLIQUE INDUZIDA BALANCEADA MÁXIMA

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

Orientador: Dr. Wladimir Araújo Tavares

Co-Orientador: Dr. Fábio Carlos Sousa Dias

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)

S578a Silva, Alan Martins.
Um algoritmo exato para o problema da biclique balanceada induzida máxima / Alan Martins Silva. – 2016.
39 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Sistemas de Informação, Quixadá, 2016.
Orientação: Prof. Dr. Wladimir Araujo Tavares.
Coorientação: Prof. Dr. Fábio Carlos Sousa Dias.

1. Grafos bipartidos. 2. Algoritmos. 3. Branch and bound. I. Título.

CDD 005

ALAN MARTINS SILVA

UM ALGORITMO EXATO PARA O PROBLEMA DA BICLIQUE INDUZIDA BALANCEADA MÁXIMA

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

Aprovada em:

BANCA EXAMINADORA

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

Dr. Fábio Carlos Sousa Dias (Co-Orientador)
Campus Quixadá
Universidade Federal do Ceará - UFC

Mr. Paulo Henrique Macedo de Araujo
Campus Quixadá
Universidade Federal do Ceará - UFC

Aos meus pais, meus amigos e meus professores.

AGRADECIMENTOS

Ao meu pai por todo o suporte que me foi dado durante todos os anos de estudo e confiança depositada. A minha vó por estar sempre disponível quando precisei.

Ao Prof. Dr. Wladimir Araújo Tavares, pela excelente orientação, suporte e ajuda durante o desenvolvimento do projeto.

Aos amigos que me acompanharam, me ajudaram e fizeram parte do meu percurso pela graduação.

Aos professores participantes da banca examinadora pelo tempo, pelas valiosas colaborações e sugestões.

"Stay Hungry, Stay Foolish"

(Whole Earth Catalog)

RESUMO

Nesta monografia discutiremos o problema da biclique induzida balanceada máxima, algumas aplicações nos dias atuais e o estado da arte do problema. Além disso, apresentaremos um algoritmo exato baseado no método da Bonecas Russas para o problema. Apresentaremos uma comparação com o algoritmo de Branch-and-Bound do estado da arte apresentado em (MCCREESH; PROSSER, 2014).

Grafos Bipartidos. Algoritmos. Branch and Bound

ABSTRACT

In this undergraduated thesis, we will discuss the Maximum Balanced Induced Biclique Problem, some applications in the present day and the state of the art of the problem. In addition, we will present an exact algorithm based on the Russian Dolls method for the problem. We will present a comparison with the state-of-the-art Branch-and-Bound algorithm presented in (MCCREESH; PROSSER, 2014).

Keywords: Graph. Biclique. Russian Dolls. Bitset

LISTA DE FIGURAS

LIST OF ALGORITHMS

SUMÁRIO

1 INTRODUÇÃO

Um grafo é uma estrutura matemática natural para modelar relações entre dados. A detecção de padrões nos relacionamentos entre os dados não é uma tema de interesse puramente acadêmico, ele possui um longo histórico de aplicações relevantes. Um exemplo interessante emerge no contexto de redes sociais. Um usuário de uma rede social não tem apenas amigos, mas ele possui características específicas de perfis: filmes que assistiu, livros que leu e grupos de usuários que ele pertence. A relação entre os usuários e suas características são capazes de descrever um perfil em uma rede social. Essas informações são extremamente relavantes para sistemas de recomendação de características ou usuários. Também podemos modelar a relação dos genes com um subconjunto de condições (por exemplo, condições que representam doenças). Dessa maneira, podemos descobrir os genes mais relacionados com certos tipos de doenças. Esses tipos de informações podem ser extremamente úteis para o desenvolvimento de novos fármacos.

O foco deste trabalho será na detecção de uma subestrutura densa fundamental chamada *biclique*. Uma *biclique*, ou subgrafo bipartido completo, é um par de subconjuntos disjuntos {A,B} tal que existe uma aresta ligando todo vértice de A para todo vértice de B. Por exemplo, considere um grafo relacionado com as redes sociais, onde existem dois tipos de vértices, usuários e filmes. Existe uma aresta entre um usuário e um filme se o usuário avaliou positivamente o filme. Uma biclique no grafo consiste em um conjunto de usuários U e um conjunto de filmes F tal que todo usuário de U gostou de todo filme de F. Essa informação é extremamente importante para entender e predizer os próximos filmes que ele pode gostar. Uma biclique é balanceada se $|U| = |F|$, e induzida se não existe uma aresta entre dois vértices de U e dois vértices de F. O problema da biclique induzida balanceada máxima é encontrar uma biclique induzida balanceada de maior cardinalidade em um grafo arbitrário.

O problema da biclique induzida balanceada máxima em grafos arbitrários é NP-Díficil (MCCREESH; PROSSER, 2014). A menos de P = NP, não se conhece uma solução polinomial para o problema. Em (MCCREESH; PROSSER, 2014), um algoritmo de branch-and-bound combinatório é apresentado para o problema combinando diversas técnicas.

Em (TRUKHANOV et al., 2013), um framework algorítmico de propósito geral exato baseado no método das bonecas russas para resolver problema de encontrar um subconjunto de vértices máximo satisfazendo um propriedade Π interessante, não trivial, que seja hereditária nos subgrafos induzidos. O algoritmo é adaptado para resolver o problema do k-plex máximo,

clique s-deficiente máxima.

Esse mesmo framework foi aplicado anteriormente para o problema da clique máxima (ÖSTERGåRD, 2002), problema da clique máxima ponderada (ÖSTERGåRD, 1999), problema da cobertura tripla de Steiner (ÖSTERGåRD; VASKELAINEN, 2011), problema do subtorneio transitivo máximo (KIVILUOTO; ÖSTERGåRD; VASKELAINEN, 2014) e o problema do melhor churrasco combinatório (ÖSTERGåRD; VASKELAINEN; MOSIG, 2007).

Em (SHAHINPOUR, 2013), um algoritmo para o problema da biclique máxima é apresentado utilizando o framework apresentado por (TRUKHANOV et al., 2013). Vale salientar que o problema da biclique máxima cumpre as propriedades para a utilização do framework, enquanto o problema da biclique máxima balanceada não.

O algoritmo do estado da arte para o problema da biclique induzida balanceada máxima é o algoritmo proposto por (MCCREESH; PROSSER, 2014). Este algoritmo utiliza várias técnicas, dentre as principais, podemos citar:

1. Limite superior baseado na partição em cliques;
2. Estratégia de ramificação em largura baseada na partição de cliques;
3. Paralelismo de bits;
4. Alternância de paramêtros durante as chamadas recursivas.
5. Ordem estática dos vértices.

Nesta monografia, apresentaremos um algoritmo exato de bonecas russsas utilizando algumas técnicas apresentadas em (MCCREESH; PROSSER, 2014). No capítulo 2, apresentaremos alguns trabalhos relacionados, no capítulo 3, é apresentada a fundamentação teórica necessária para a leitura do texto, no capítulo 4 teremos os materiais e métodos utilizados durante a criação deste trabalho, no capítulo 5, serão exibidos os algoritmos e resultados obtidos no decorrer do trabalho, no capítulo 6 discutiremos sobre os resultados gerais do trabalho e no capítulo 7 estão escritas as considerações finais.

2 TRABALHOS RELACIONADOS

2.1 Algoritmo de Branch-and-Bound

Em (TOMITA; SEKI, 2003) é desenvolvido um algoritmo de Branch-and-Bound para o problema de encontrar a maior clique em um grafo arbitrário. No trabalho é utilizada uma técnica de coloração aproximada como crítério de corte nas ramificações do algoritmo. Os resultados mostraram que o algoritmo desenvolvido por (TOMITA; SEKI, 2003) possuia um desempenho bem melhor que o algoritmo de estado da arte daquele período.

Em (MCCREESH; PROSSER, 2014), é apresentado um algoritmo de branch and bound combinatório utilizando as seguintes técnicas:

1. Procedimento para quebra de simetria obtido através da alternância da passagem dos parâmetros nas chamadas recursivas;
2. Procedimento de limite superior baseado em uma heurística de partição em cliques;
3. Uma ordem estática dos vértices para a construção das partições em cliques. Essa ordem estática evita que a reordenação dos vértices seja executada toda vez que a heurística de partição de cliques é realizada;
4. Estratégia de ramificação baseada em uma ordem induzida pelas partições em cliques.

Além desses componentes, em (MCCREESH; PROSSER, 2014) é utilizado uma estrutura de dados, usada para representar a adjacência de cada vértice, chamada de vetor de bits. Os vetores de bits são estruturas de dados capazes de realizar o mapeamento de cada bit individualmente. Esse tipo de estrutura de dados tem duas vantagens básicas:

- Permite uma redução na memória utilizada na representação;
- Paralelizar certas operações realizadas durante o algoritmo. Esse tipo de paralelismo é chamado de paralelismo de bits. A utilização desta ténica pode ser encontrada em diversos problemas como (MCCREESH; PROSSER, 2014), (SEGUNDO et al., 2013) e (CORRÊA et al., 2014).

Comparamos os resultados computacionais do nosso trabalho com os resultados obtidos em (MCCREESH; PROSSER, 2014), tornando possível, assim, visualizar de forma comparativa a performance do nosso trabalho em comparação com o estado da arte do problema.

2.2 Bonecas Russas

O método das Bonecas Russas foi proposto originalmente em (VERFAILLIE; LEMAÎTRE; SCHIEX, 1996) para problemas de satisfação de restrições. De maneira independente, Ostergaard aplica o método ao problema da clique ponderada em (ÖSTERGåRD, 1999).

Em (VASKELAINEN et al., 2010), fica claro que, ao modificarmos um algoritmo já existente, inserindo a técnica das Bonecas Russas, pode ser possível obter resultados melhores quando comparado ao algoritmo original.

Também utilizado em (SANCHEZ et al., 2009) para problemas de processamento de modelos gráficos e em (VASKELAINEN et al., 2010) para o problema de *Steiner triple covering*, o método das bonecas russas se mostrou útil na obtenção de bons resultados e foi recomendado para aplicação em outros problemas pelos autores de ambos os artigos.

(TRUKHANOV et al., 2013) estende a aplicação do método para resolver problemas relacionados a encontrar um subconjunto máximo satisfazendo uma propriedade que seja hereditária nos subgrafos induzidos. O framework é adaptado para duas propriedades hereditárias em grafos k-plex e clique s-deficiente.

Em (SHAHINPOUR, 2013), o método é aplicado ao problema da biclique induzida (uma propriedade hereditária nos subgrafos induzidos).

Neste trabalho, mostramos que mesmo quando uma propriedade não é hereditária nos subgrafos induzidos, como é a biclique induzida balanceada, é possível desenvolver um algoritmo de bonecas russas.

(VERFAILLIE; LEMAÎTRE; SCHIEX, 1996) deixa claro que inserir um limite superior melhor do que já está sendo utilizado pode possibilitar um desempenho melhor quando comparado ao mesmo algoritmo utilizando apenas um limite superior.

3 FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão apresentados conceitos sobre grafos, algoritmos de aproximação e técnicas utilizadas em algoritmos relacionados a grafos. A seguir serão explicados os conceitos necessários para entendimento deste trabalho. As linhas com bordas retangulares nos algoritmos deste trabalho são os locais onde é possível utilizar o paralelismo de bits.

A fundamentação teórica tem como base as explicações presentes em (ÖSTERGåRD, 2002), (TOMITA et al., 2010), (VERFAILLIE; LEMAÎTRE; SCHIEX, 1996), (SANCHEZ et al., 2009) e (VASKELAINEN et al., 2010), onde muitos dos conceitos que se aplicam a este trabalho são explicados e utilizados.

3.1 Grafos

Um **grafo simples** G é um par ordenado (V,E) composto por um conjunto finito de vértices $V = \{v_1, v_2, \ldots, v_{|V|}\}$ e um conjunto de arestas $E \subseteq \{(u,v) : u,v \in V, u \neq v\}$. Para todo grafo G, denotamos $V(G)$ e $E(G)$, respectivamente, os conjuntos de vértices e o conjunto de arestas de G. Os grafos utilizado neste trabalho serão grafos simples e não direcionados, ou seja, a aresta (u,v) é igual a aresta (v,u). Na Figura 1, podemos ver um exemplo desse tipo de grafo:

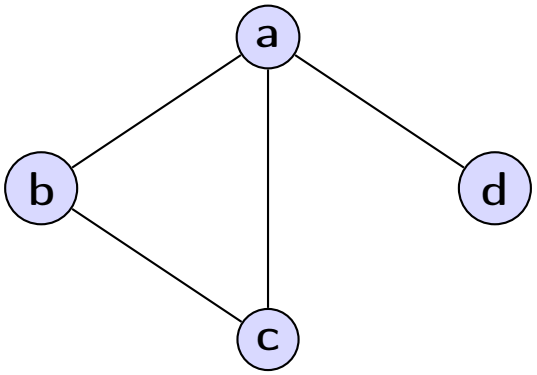

Figura 1 – Grafo G(V, E), onde V = {a, b, c, d} e E = {(a, b), (a, d), (a, c), (b, c)} - Fonte: Próprio autor

Um grafo H é dito um **subgrafo** de G, se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Um grafo é dito um **subgrafo induzido** por S, denotado por $G[S]$, se $G[S] = (S, E(G) \cap (S \times S))$. Na Figura 2, temos um exemplo de um subgrafo induzido H de G.

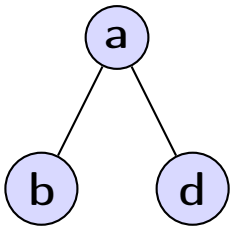

Figura 2 – Grafo H subgrafo do grafo G da Figura 1 - Fonte: Próprio autor

Um conjunto independente é um conjunto de vértices onde não há aresta conectando dois vértices daquele conjunto. Conjuntos independentes disjuntos é um grupo de conjuntos independentes em que a interseção de quaisquer dois conjuntos independentes do grupo é vazia.

Um grafo é chamado de um grafo k-partido se V pode ser particionado em k conjuntos independentes disjuntos. Quando $k = 2$, o grafo $k - partido$ é chamado de bipartido. Um grafo bipartido pode ser denotado da seguinte maneira $G = (A \cup B, E)$, onde A e B são conjuntos independentes, $E \subseteq \{\{u,v\} : u \in A, v \in B\}$. Na Figura 3, temos um exemplo de grafo bipartido $G(A \cup B, E)$.

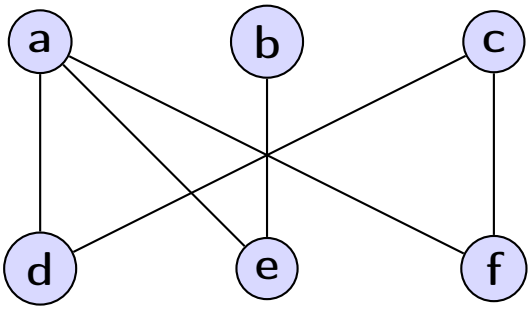

Figura 3 – Grafo bipartido $G = (A \cup B, E)$, onde $A = \{a,b,c\}$,$B = \{f,d,e\}$ e $E = \{\{c,f\},\{c,d\},\{a,f\},\{a,d\},\{a,e\},\{b,e\}\}$ Fonte: Próprio autor

Uma **clique** C de G é um subgrafo de G tal que todo par de vértice de C é adjacente entre si, ou seja, $\forall u, v \in C, u \neq v \wedge \{u,v\} \in E(G)$. Na Figura 4, temos um exemplo de uma clique C de G com cardinalidade 3.

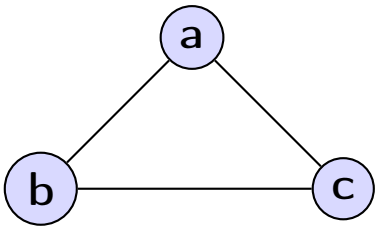

Figura 4 – Exemplo de Clique - Fonte: Próprio autor

Um par de subconjuntos disjuntos A e B de V é chamada uma **biclique** C de G se existe uma aresta $(a,b) \in E(G)$, para todo $a \in A$ e $b \in B$. Dessa maneira, o conjuntos de aresta de C forma um **subgrafo bipartido completo**, também chamado de **biclique** de G. Uma biclique $C = A \cup B$ será induzida se A e B são conjuntos independentes. Na Figura 5, podemos ver uma bicliqe induzida $\{a,b\} \cup \{c,d,e\}$.

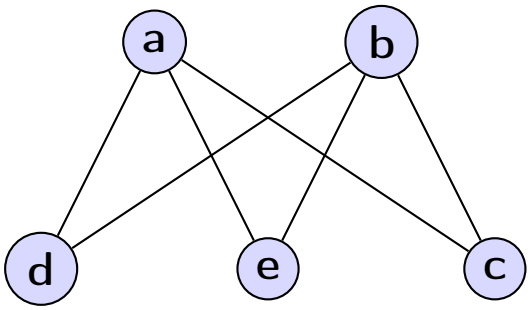

Figura 5 – Exemplo de biclique com A = {a, b} e B = {c, d, e} - Fonte: Próprio autor

Uma biclique é dita **balanceada** quando $|A| = |B|$. Veja Figura 6.

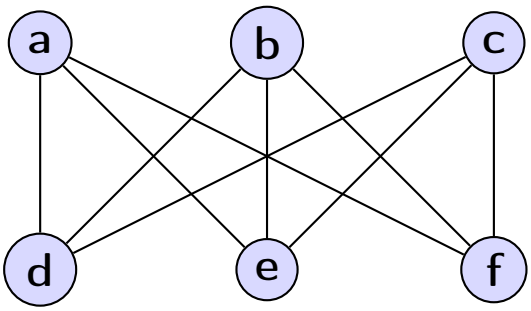

Figura 6 – Exemplo de biclique balanceada com A = {a, b, c} e B = {d, e, f} - Fonte: Próprio autor

Uma biclique balanceada C^* de G é dita **máxima biclique balanceada** se não existe uma biclique balanceada C de G, onde $C^* \neq C$ e $|C| > |C^*|$.

Definimos o tamanho da clique máxima de um grafo G como $\omega(G)$. A partição em cliques é o ato de particionar os vértices de um grafo em conjuntos, onde cada conjunto é uma clique. Em (MCCREESH; PROSSER, 2014) é introduzido o símbolo $\ddot{\omega}(G)$ para definir o tamanho da biclique máxima induzida (|A| + |B|).

Uma partição do conjunto A é uma coleção $A_1, A_2, \ldots, A_n$ de subconjuntos não vazios tal que $\bigcup_{i=1}^{n} A_i = A$ e $A_i \cap A_j = \emptyset$, para todo $i \neq j$.

Uma partição em cliques de G é uma partição do conjunto de vértices $V(G)$ em cliques $K_1, K_2, \ldots, K_m$. O menor número de cliques de uma partição em cliques é denotado $\theta(G)$. O menor número de clique de uma partição em cliques é um limite superior para $\ddot{\omega}(G)/2$.

$$\ddot{\omega}(G)/2 \leq \theta(G) \tag{3.1}$$

3.2 Algoritmo de (MCCREESH; PROSSER, 2014)

O algoritmo de (MCCREESH; PROSSER, 2014) é o algoritmo de estado da arte e também o algoritmo base utilizado na nossa implementação de uma solução para o problema. O algoritmo será detalhado neste capítulo e posteriormente comparado ao nosso algoritmo.

Basicamente, um algoritmo de Branch-and-Bound é um algoritmo de enumeração implícita. As estratégias de poda permitem que vários subproblemas possam ser implicitamente enumerados sem prejudicar a obtenção da solução ótima.

Um subproblema pode ser definido por (A, B, PA, PB, LB), onde A e B definem a biclique atual já construída (não necessariamente balanceada), PA representa os vértices que podem entrar na partição A e PB os vértices que podem entrar na partição B e LB é o tamanho da maior biclique encontrada dividida por 2, ou seja, o tamanho de um dos lados da biclique balanceada.

O seguinte teorema define uma condição trivial para a poda de um subproblema.

Teorema 1. *Seja (A, B, PA, PB, LB) um subproblema da biclique balanceada máxima. Se $|A| + |PA| \leq LB$ ou $|B| + |PB| \leq LB$ então não existe nenhuma biclique balanceada (A^*, B^*) com $|A^*| = |B^*| > LB$ tal que $A^* \setminus A \subseteq PA$ e $B^* \setminus B \subseteq PB$.*

Demonstração. Suponha por absurdo que $|A| + |PA| \leq LB$ ou $|B| + |PB| \leq LB$ e que existe uma biclique balanceada $|A^*| = |B^*| > LB$. Por hipótese, $|A^*| > LB$. Como $A^* \setminus A \subseteq PA$, temos que $|A^*| \leq |A| + |PA|$. Logo,

$$LB < |A*| <= |A| + |PA|$$

□

Contradizendo o que foi assumido inicialmente.

Algorithm 1 Algoritmo de Branch-and-Bound Simples

```
1:  function BICLIQUE(G = (V,E))
2:      (A_max, B_max) ← (∅, ∅)
3:      expand(G, ∅, ∅, V(G), V(G), A_max, B_max)
4:      return (A_max, B_max)
5:  end function
6:  function EXPAND(G, A, B, PA, PB, A_max, B_max)
7:      for v ∈ PA do
8:          if |A| + |PA| > |A_max| e |B| + |PB| > |B_max| then
9:              A ← A ∪ {v}
10:             PA ← PA \ {v}
11:             PA' ← PA ∩ N̄(v)
12:             PB' ← PA ∩ N(v)
13:             if |A| = |B| e |A| > |A_max| then
14:                 (A_max, B_max) ← (A, B)
15:             end if
16:             expand(G, B, A, PB', PA', B_max, A_max)
17:             A ← A \ {v}
18:         end if
19:     end for
20: end function
```

O algoritmo 1 apresentado em (MCCREESH; PROSSER, 2014) foi a base para o desenvolvimento da solução desenvolvida no artigo e, seu funcionamento, apesar de simples, possui uma diferença na ramificação dos subproblemas.

Em subproblema (A, B, PA, PB, LB), cada vértice de $v \in PA$ dá origem um novo subproblema. Esse subproblema vai enumerar todas as bicliques induzidas balanceada contendo o vértice v. Depois disso, o vértice v pode ser removido de PA. Note que não vamos considerar os vértices que podem entrar no conjunto B. Isso é possível pois a chamada do algoritmo

recursivo realiza a alternância dos grupos, por exemplo, na execução do subproblema $(A, B, PA, PB, A_{max}, B_{max})$, a chamada recursiva será para $(B, A, PB, PA, B_{max}, A_{max})$, assim, os grupos são solucionados alternadamente, fazendo com que não seja aumentado apenas um dos lados sem que seja possível aumentar o outro.

Algorithm 2 Algoritmo de cliqueSort

```
1:  function CLIQUESORT(G, P)
2:      bounds ← vetor de inteiros
3:      order ← vetor de inteiros
4:      P' ← P
5:      k ← 1
6:      i ← 1
7:      while P' ≠ ∅ do
8:          Q ← P'
9:          while Q ≠ ∅ do
10:             v ← primeiro elemento de Q
11:             P' ← P' \ v
12:             Q ← Q ∩ N(G, v)
13:             bounds[i] ← k
14:             order[i] ← v
15:             i ← i + 1
16:         end while
17:         k ← k + 1
18:     end while
19: end function
```

A função *cliqueSort* produz dois vetores. O vetor *bounds* contem os limites do tamanho do maior conjunto independente: o subgrafo induzido pelos vértices *1* até *n* do *order* não pode possuir um conjunto independente maior que o valor de *bounds[n]*. O vetor *order* contém os vertices de *P* em uma ordem que é utilizada da direita para a esquerda, repetidamente removendo o valor mais a direita e ramificando o valor do vértice v.

Esses vetores são construidos na função *cliqueSort* da seguinte forma: A variável *P* guarda quais vértices ainda não foram alocados para uma clique e inicialmente (linha 4) contém todos os vértices do parâmetro *P*. Enquanto existirem vertices não alocados (linha 7), de forma gulosa é construída uma nova clique. A variável *Q* (line 9) guarda os vértices que podem ser adicionados a essa nova clique . Na linha 10 é selecionado um vértice v $\in$ Q, adicionado v à clique e na linha 12 são removidos de *Q* quaisquer vértices que não são adjacentes a *v* (assim todos os vértices em *Q* serão adjacentes aos vértices existentes na nova clique). Continuam-se adicionando vértices à nova clique (isto é, adicionando-se vértices ao vetor *order* com sua

respectiva clique adiciona ao vetor *bounds*) até que Q esteja vazio (linhas 13 e 14), indicando que não é possível continuar. Nesse ponto, uma nova clique é criada (linha 17 voltando para a linha 7) e é verificado se existem vértices ainda não alocados.

O seguinte teorema define uma condição de poda baseado na partição de cliques gerada pela função *cliqueSort*.

Teorema 2. *Seja* (A,B,PA,PB,LB) *um subproblema da biclique balanceada máxima. Seja bounds e orders vetores produzidos pela função* $cliqueSort(G,PA)$. *Se* $bounds[i]+|A| \leq LB$ *então não existe uma biclique balanceada* (A^*,B^*) *tal que* $|A^*| > LB$ *e* $A^* \setminus A \subseteq \{orders[1], orders[2], \ldots, orders[i]\}$.

Demonstração. Suponha por absurdo que $bounds[i]+|A| \leq LB$ e que existe uma biclique balanceada (A^*,B^*) tal que $|A^*| > LB$ e $A^* \setminus A \subseteq \{orders[1], orders[2], \ldots, orders[i]\}$. Seja $m = bounds[i]$. Então $\{orders[1], orders[2], \ldots, orders[i]\}$ pode ser particionado em $K_1, K_2, \ldots, K_m$ cliques. $A^* \setminus A$ pode conter no máximo um vértice de cada clique K_i.

$$\begin{aligned} |A^* \setminus A| &\leq m \\ |A^*| - |A| &\leq bounds[i] \\ |A^*| &\leq bounds[i] + |A| \leq LB \\ |A^*| &\leq LB \end{aligned}$$

Contradizendo o que foi assumido inicialmente. □

Algorithm 3 Algoritmo de ordenação inicial dos vértices

```
1:  function ORDENACAO(G)
2:      π ← vetor de inteiros
3:      usado ← vetor de booleanos
4:      grau ← vetor de inteiros
5:      delta ← 0
6:      for v ∈ V(G) do
7:          grau[v] ← 0
8:          for u ∈ V(G) do
9:              if (u,v) ∈ E(G) then
10:                 grau[v] ← grau[v] + 1
11:                 if grau[v] > delta then
12:                     delta ← grau[v]
13:                 end if
14:             end if
15:         end for
16:     end for
17:     count ← |V(G)|
18:     repeat
19:         min ← delta + 1
20:         p ← −1
21:         for v ∈ V(G) do
22:             if ¬usado[v] ∧ grau[v] < min then
23:                 min ← grau[v]
24:                 p ← v
25:             end if
26:         end for
27:         π[count] ← p
28:         usado[p] ← true
29:         count ← count − 1
30:         for u ∈ V(G) do
31:             if ¬usado[u] ∧ u ≠ p ∧ (p,u) ∈ E(G) then
32:                 grau[u] ← grau[u] − 1
33:             end if
34:         end for
35:     until count ≥ 1
        return π
36: end function
```

O Algoritmo *ORDENAÇÃO* retorna um vetor de inteiros com os vértices ordenados em critério de grau. Inicialmente são criados 3 vetores e uma variável inteira(linhas 2 a 5), π é onde ficarão armazenados os vértices ordenados, *usado* armazena se o vértice já foi ou não processado(i foi processado se $usado[i]$ é verdade), *grau* armazena o grau do vértice(o grau do vértice i é $grau[i]$) e a variável $delta$ armazena o maior grau encontrado. Processando todos os vértices do grafo(linha 6), primeiramente atribui-se o grau 0 para aquele vertice(linha 7), depois,

para cada vizinhos de v(linhas 8 e 9), seu grau é aumentado em 1(linha 10) e se o grau atingido se torna o maior já encontrado, o valor é armazenado em $delta$(linhas 11 e 12).

Agora, é necessário ordenar esses vértices, e para indicar quantos vértices ainda faltam ser ordenados, utilizamos *count* inicialmente com o número total de vértices e vamos decrementando o valor em 1 para cada vértice ordenado.(linha 17)

Repetidamente, cria-se *min* para com $infinito$(se o valor máximo de grau é $delta$, um grau $delta+1$ nunca será encontrado). O vértice a ser ordenado em cada execução é armazenado em p. Para todos os vértices de G, se o vértice não foi processado ainda e possui um grau menor que o menor valor já encontrado(linhas 21 e 22), o vértice atual pode ser elegido como vértice com menor grau, atualizando p e *min*(linhas 23 e 24). Após verificar todos os vértices de G, p possui o vértice com menor grau encontrado, assim, adiciona-se o vértice ao fim de π(linha 27), marca-se aquele vértice como já processado(linha 28) e o *count* é decrementado(linha 29). Com o vértice p já processado, os vértices na vizinhança de p tem seu grau decrementado em 1 unidade(linhas 30 a 32). Quando não existem mais vértices a serem processados(linha 35), pode-se retornar π.

Algorithm 4 Algoritmo de reconstrução de adjacência

```
1:  function RECONSTRUÇÃOADJACÊNCIA(G, π)
2:      for i ← 1 até |V| do
3:          N_G(π[i]) ← ∅
4:          for j ← 1 até |V| do
5:              if (π[i], π[j]) ∈ E then
6:                  N_G(π[i]) ← N_G(π[i]) ∪ {π[j]}
7:              end if
8:          end for
9:      end for
10: end function
```

O algoritmo de reconstrução de adjacência reconstrói a vizinhança de um grafo G a partir dos vértices ordenados em um vetor(neste trabalho, o vetor produzido por *ORDENAÇÃO*).

Para os vértices em ordem de π, iniciamos a sua vizinhança com $\emptyset$(linhas 2 e 3). Após reiniciar a vizinhança do vértice $\pi[i]$, novamente percorrendo o vetor em ordem, se existia uma aresta entre os vértices $\pi[i]$ e $\pi[j]$ no grafo original(linhas 4 e 5), $\pi[j]$ é adicionado ao conjunto que representa a vizinhança de $\pi[i]$(linha 6).

3.2.1 Algoritmo base

O algoritmo de Branch-and-Bound desenvolvido em (MCCREESH; PROSSER, 2014) é o seguinte:

Algorithm 5 Algoritmo de (MCCREESH; PROSSER, 2014)

```
1:  function BICLIQUEMELHORADA(G)
2:      (A_max, B_max) ← (∅, ∅)
3:      π ← ORDENACAO(G)
4:      reconstrucaoAdjacência(G, π)
5:      expand(G, ∅, ∅, V(G), V(G), A_max, B_max)
6:      return (A_max, B_max)
7:  end function
8:  function EXPAND(G, A, B, PA, PB, A_max, B_max)
9:      (bounds, order) ← cliqueSort(G, PA)
10:     for i ← |PA| a 1 do
11:         if bounds[i] + |A| > |A_max| e |PB| + |B| > |B_max| then
12:             v ← order[i]
13:             A ← A ∪ {v}
14:             PA ← PA \ {v}
15:             PA' ← PA ∩ N_G(v)‾
16:             PB' ← PB ∩ N_G(v)
17:             if |A| = |B| e |A| > |A_max| then
18:                 (A_max, B_max) ← (A, B)
19:             end if
20:             if PB' ≠ ∅ then
21:                 EXPAND(G, B, A, PB', PA', B_max, A_max)
22:             end if
23:             A ← A \ {v}
24:             if B = ∅ then
25:                 PB ← PB \ {v}
26:             end if
27:         end if
28:     end for
29: end function
```

A execução inicial do algoritmo é com a chamada da função *BICLIQUEMELHORADA* passando o grafo como argumento.

Cada subproblema pode ser representado pela quádrupla (A, B, Pa, Pb). Os vértices que fazem parte da solução são armazenados separadamente em dois grupos A e B e os vértices candidatos são salvos em dois grupos Pa e Pb representando respectivamente os candidatos a entrar no grupo A e os candidatos a entrar no grupo B.

Durante a execução de um subproblema apenas os vértices candidatos de participar do grupo A são avaliados. Isso acontece por que na chamada recursiva os grupos são passados com a ordem invertida, trocando A por B e Pa por Pb, assim, um subproblema (A, B, Pa, Pb) vai executar recursivamente o problema (B, A, Pb', Pa').

O algoritmo inicia utilizando a função *cliqueSort* e segue a ordem recebida para testar as possíveis soluções(linha 2). Percorrendo a ordem do último para o primeiro(linha 3), se é possível obter uma solução melhor que a melhor já encontrada(linha 4), escolhe-se o vértice *order[i]*, considera-se como parte da solução, atualizam-se e criam os novos conjuntos candidatos(linhas 6 até 9). Após isso, se foi encontrada uma solução melhor, salva-se a solução(linha 11) e se ainda existem candidatos para entrar no conjunto B da solução, a função recursiva é executada. Quando o algoritmo prosseguir, remove-se *order[i]* da solução (linha 16) e para evitar simetria, se $B = \emptyset$, significa que é a primeira execução do algoritmo(é como testar se o algoritmo está na raiz da árvore de ramificação), assim, podemos remover *order[i]* dos candidatos a entrar no grupo B pois testá-lo iria gerar um resultado simétrico ao teste de com $v \in A$.

Segundo (MCCREESH; PROSSER, 2014), se podemos cobrir um Grafo G com k cliques, não é possível obter um conjunto independente maior que k, pois, cada elemento de um conjunto independente deve estar em uma clique diferente. O problema dividir um grafo em cliques é NP-completo, porém, é possível utilizar o algoritmo guloso *cliqueSort* para encontrar um resultado razoável em tempo polinomial. O algoritmo de cliqueSort implementado vem de (TOMITA et al., 2010), assim como a afirmação de que utilizar o resultado do *cliqueSort* como limite superior é melhor do que utilizar o próprio tamanho de *Pa*.

O procedimento de quebra de simetria ocorre nas linhas 15 e 16 da Figura 6. Se o grupo B estiver vazio, baseado no modo alternado de preenchimento dos grupos, é correto afirmar que essa é a primeira execução do algoritmo, assim, se o vértice *v* foi escolhido e considerado em A como primeira opção, testá-lo em B em algum momento seria criar uma solução simétrica, em que, no pior caso, levaria ao gasto do dobro de tempo para ser calculada. Removê-la dos candidato a entrar no grupo B resolve o problema de simetria de resultados.

3.3 Paralelismo de bits

Também utilizado em (SEGUNDO et al., 2013) e (CORRÊA et al., 2014), a técnica de paralelismo de bits consiste em armazenar as informações em forma de mapa de bits de

forma a permitir a utilização de operações de bit paralelo. Um mapa de bits é uma estrutura de dados para codificação de conjuntos que armazenam elementos individual do conjunto de forma compacta enquanto permite acesso direto ao endereço de cada elemento (CORRÊA et al., 2014).

O mais interessante é a sua habilidade de se beneficiar de potenciais operações a nivel de bit disponíveis em hardware para executar operações de conjunto coletivas através de rápidas operações de máscara de bits(interseção de dois conjuntos é um exemplo típico) (CORRÊA et al., 2014).

As operações bit paralelo são utilizadas neste trabalho e em (MCCREESH; PROSSER, 2014) em operações de interseção($w \cap u$) e união($w \cup u$) de conjuntos.

3.4 Implementação do algoritmo

Tabela 1 – Comparação do algoritmo original e o implementado por nós

Instância	**MCCREESH** original		**MCCREESH** implementado	
	#subproblemas	tempo(s)	#subproblemas	tempo(s)
*C*500.9	107553	0.072	107636	0.088
*C*1000.9	7358668	6.183	7336258	**5.989**
*MANN_a*27	1407	0.001	1409	0.004
*MANN_a*45	9852	0.017	9854	0.019
*MANN_a*81	53902	0.336	53904	**0.174**
*brock*200_1	57931	0.018	55494	0.033
*brock*200_4	102484	0.029	105727	0.049
*brock*400_1	1774969	0.855	1775557	1.051
*brock*400_1	1774969	0.855	1775557	1.051
*brock*400_4	1772650	0.834	1790335	1.063
*gen*200_*p*0.9_44	2628	0.001	2729	0.005
*gen*200_*p*0.9_55	4201	0.003	3999	**0.002**
*p_hat*5002	5881265	2.206	5894856	2.908
*p_hat*5003	6438257	3.205	6397965	3.832
*sanr*400_0.5	13683397	4.706	11323056	5.137
*keller*4	82646	0.025	62638	0.034
*johnson*1624	2216795	0.135	1163598	0.166
*sanr*400_0.7	3370144	1.552	2947956	1.691

Os testes estão sendo realizados executando a implementação disponibilizada pelo próprio autor. Infelizmente, a técnica utilizada para ordenar os vértices não é descrita em (MCCREESH; PROSSER, 2014), assim, ao implementar o algoritmo 5, optamos por utilizar a ordenação descrita no algoritmo 3.

Os resultados da comparação do algoritmo 5 disponibilizado pelo autor e o algoritmo 5 implementado por nós são mostrados na tabela 3.

Apesar de bem similares, os resultados possuem diferenças em número de subproblemas resolvidos. Acredita-se que essa diferença é causada pela utilização de algoritmo de ordenação distintos.

4 ALGORITMO PROPOSTO

4.1 Método das Bonecas Russas

No algoritmo 6 descrevemos o algoritmo de Bonecas Russas de (TRUKHANOV et al., 2013) que foi utilizado como modelo neste trabalho.

Algorithm 6 Algoritmo de Bonecas Russas de (TRUKHANOV et al., 2013)

```
1:  function BONECASRUSSAS(G, Π)
2:      ordenacao(V)
3:      max ← 0
4:      for i ← 1 até n do
5:          C ← {v ∈ S_i \ {v_i} : {v, v_i} que satisfaz Π}
6:          v ← V[i]
7:          FINDMAXΠ(C, {v_i})
8:          μ[i] ← max
9:      end for
10:     return max
11: end function
12: function FINDMAXΠ(C, P)
13:     if |C| > max then
14:         max ← |C|
15:     end if
16:     while P ≠ ∅ do
17:         if |C| + |P| < max then return
18:         end if
19:         i ← min{j : v_j ∈ P}
20:         if μ[i] + |P| < max then return
21:         end if
22:         P ← P \ {v_i}
23:         C' ← C ∪ {v_i}
24:         P' ← {v ∈ P : C' ∪ {v} satisfaz Π}
25:         FINDMAXΠ(C', P')
26:     end while
27: end function
```

No algoritmo 6, algumas variáveis (como max) são compartilhadas com a função $FINDMAX\Pi$. Para cada subproblema resolvido (retorno da função chamada na linha 7), o valor de *max* foi alterado (dentro da função $FINDMAX\Pi$) e possui o valor máximo de solução encontrado para o subproblema i, com isso, essa solução é armazanada em μi (linha 8) e esse valor será utilizado como limite superior.

A função $FINDMAX\Pi$ vai resolver cada boneca gerada em *BONECASRUSSAS*.

Primeiro, é checado se a solução atual ultrapassou a melhor solução encontrada, se sim, o valor da melhor solução é alterado (linhas 13 e 14). Enquanto existirem candidatos a entrar na solução (linha 16), fazem-se os cortes existentes ou é processado o subpproblema. O primeiro corte (linha 17) ocorre quando não é possível, com o tamanho da solução atual somado com o número de candidatos, ultrapassar o valor máximo já encontrado. O segundo corte é o corte proporcionado pela Boneca Russa, onde, ao selecionar um vértice do conjunto candidato (linha 19), se a melhor solução que pode ser encontrada com aquele vértice($\mu[i]$) somada com o número de candidatos não consegue ser maior que a melhor solução já encontrada, o subproblema é podado(linha 20). Por fim, remove-se v_i do conjunto candidato(linha 22), adiciona-se v_i no conjunto de solução(linha 23), cria-se o novo conjunto candidato(linha 24) e a chamada recursiva ocorre(linha 25).

Abaixo temos a implementação da técnica das bonecas russas desenvolvida para o problema em questão. Denotamos a instância de grafo G, os dois conjuntos do resultado como A e B. A execução incial do algoritmo é *BonecasRussas(G, ∅, ∅)* com A e B vazios.

Algorithm 7 Algoritmo de Bonecas Russas desenvolvido

```
 1: function BONECASRUSSAS(G, A, B)
 2:     π ← ORDENACAO(G)
 3:     RECONSTRUCAOADJACENCIA(G, π)
 4:     R ← vetor de inteiros
 5:     for i ← 1 até |V_G| do
 6:         v ← V_G[i]
 7:         PA ← V_G \ N_G(v)
 8:         PB ← V_G \ N_G(v)
 9:         A' ← {v}
10:         B' ← ∅
11:         NOVOEXPAND(G, B', A', PB, PA, B, A, R)
12:         R[v] ← |A|
13:     end for
14: end function
```

Quando comparado com o algoritmo de (TRUKHANOV et al., 2013), nosso algoritmo funciona de maneira similar. Primeiramente, o conjunto de vértices é ordenado (linha 3) e um conjunto *R* onde as melhores solução para cada subproblema serão armazenadas. A sequência utilizada para percorrer *V* é da direita para a esquerda (linha 5). Para cada subproblema, criam-se seus conjuntos candidatos e considera-se que o vértice *v* faz parte da solução (linhas 8 a 10). Após a chamada da função *EXPAND* (linha 12), |A| possui a melhor

solução encontrada para o subproblema e esse valor pode ser armazenado em R.

Em seguida, apresentamos o algoritmo utilizado para resolver cada boneca.

Algorithm 8 Algoritmo expand desenvolvido

```
1:  function NOVOEXPAND(G, A, B, PA, PB, A_max, B_max, R)
2:      if |A| = |B|e|A| > |AM| then
3:          (AM, BM) ← (A, B)
4:      end if
5:      while |A| + |PA| > |A_max|e|B| + |PB| > |B_max| do
6:          v ← last(PA)
7:          if max(|A|, |B|) + R[v] ≤ |A_max| then
8:              return
9:          end if
10:         A' ← A ∪ {v}
11:         PA ← PA \ {v}
12:         PA' ← PA \ N_G(v)
13:         PB' ← PB \ N_G(v)
14:         NOVOEXPAND(G, B', A', PB', PA', B_max, A_max, R)
15:     end while
16: end function
```

O funcionamento do algoritmo *NOVOEXPAND* funciona de maneira semelhante ao algoritmo *expand* de (MCCREESH; PROSSER, 2014). As diferença entre os dois é, primeiramente, em não utilizar o algoritmo de *cliqueSort*. A ordem utilizada pelo algoritmo *NOVOEXPAND* é uma ordem contrário à ordem estática de *PA* (sempre utilizando o último valor, como mostrado na linha 6). Outra diferença no algoritmo é a adição do corte das Bonecas Russas (linha 7).

Se $R[v]$ é a melhor solução encontrada para o vértice v, ao escolher v para entrar em uma nova solução, na melhor hipótese, existe a possibilidade de adicionar $R[v]$ vértices à solução. Dessa forma, a melhor solução possível é o tamanho da solução atual(maior valor entre |A| e |B|) somado com $R[v]$. Se a melhor solução possível não ultrapassa o valor da melhor solução encontrada(A_{max}), não é necessário resolver esse subproblema pois não nos dará resultados melhores.

Teorema 3. *Se a maior biclique induzida balanceada (A,B) em G for menor que k então a maior biclique induzida (A^*,B^*) em G tem $|A^*| \leq k$ ou $|B^*| \leq k$.*

Demonstração. Suponha por absurdo que a maior biclique induzida (A^*,B^*) em G tem $|A^*| > k$ e $|B^*| > k$ então existe uma biclique induzida balanceada em G com cardinalidade maior que k.

□

Se a biclique atual (A,B) for balanceada, ou seja, $|A| = |B|$. É fácil ver que se a maior biclique induzida balanceada em PA for menor que LB - max(|A|,|B|) = LB - |A| então não podemos encontrar uma biclique que supere a melhor biclique encontrada.

Se a biclique atual (A,B) for desbalanceada, ou seja, |A| > |B| ou |B| > |A| e a maior biclique induzida balanceada em PA for menor que LB - max(|A|,|B|). Pelo Lema anterior, a maior biclique induzida (|A*|,|B*|) em PA tem |A*| $\leq$ LB - max(|A|,|B|) ou |B*| $\leq$ LB - max(|A|,|B|). Logo, a maior biclique induzida balanceada em G tem cardinalidade LB - max(|A|,|B|) + max(|A|,|B|).

5 RESULTADOS

Neste capítulo vamos apresentar os resultados obtidos pelo algoritmo desenvolvido em comparação com o algoritmo de (MCCREESH; PROSSER, 2014). O algoritmo foi desenvolvido sem recursos de *multithreading*, utilizando a linguagem C++ e paralelismo de bits. Os experimentos foram realizados em um computador com Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz e 6GB de memória RAM. Os tempos computados não incluem o tempo de leitura do arquivo com os dados do grafo, mas incluem o tempo de pré-processamento e ordenação, além do tempo gasto pelo algoritmo em si.

As instâncias utilizadas foram obtidas em (DIMACS, 2016) e foram iguais para ambos os algoritmos. Os resultados da comparação estão na tabela 2.

Tabela 2 – Resultados

Instância	**MCCREESH**		**BONECAS RUSSAS**	
	#subproblemas	tempo(s)	#subproblemas	tempo(s)
*C*1000.9	7358668	4.607	268485938	26.070
*C*125.9	920	0.000	13935	0.001
*C*2000.5	*aborted*	*aborted*	*aborted*	*aborted*
*C*2000.9	318154620	377.658	7003305611	1037.018
*C*250.9	12448	0.003	181350	0.012
*C*4000.5	*aborted*	*aborted*	*aborted*	*aborted*
*C*500.9	107553	0.051	7735201	0.466
*MANN_a*27	1407	0.001	6222	0.003
*MANN_a*45	9852	0.014	28071	**0.011**
*MANN_a*81	53902	0.258	160068	**0.118**
*MANN_a*9	32	0.000	246	0.000
*brock*200_1	57931	0.013	1059983	0.047
*brock*200_2	171207	0.030	1237181	0.050
*brock*200_3	118038	0.023	1275675	0.055
*brock*200_4	102484	0.020	1111775	0.047
*brock*400_1	1774969	0.596	31585717	1.955
*brock*400_2	1760558	0.589	37219169	2.365
*brock*400_3	1812188	0.603	37222534	2.482
*brock*400_4	1772650	0.594	32342845	1.933
*brock*800_1	951765108	454.088	30661321060	2634.999
*brock*800_2	913607053	443.864	29440407129	2536.681
*brock*800_3	962767080	457.808	35242524744	2896.737
*brock*800_4	936273493	447.023	32985403695	2722.503
*c-fat*200-1	214	0.000	1606	0.001
*c-fat*200-2	353	0.000	4315	0.001
*c-fat*200-5	927	0.001	28831	0.001
*c-fat*500-1	523	0.001	4599	0.002
*c-fat*500-10	4219	0.010	277207	0.014
*c-fat*500-2	619	0.001	10569	0.002
*c-fat*500-5	1398	0.002	51031	0.004
*gen*200_*p*0.9_44	2628	0.001	34656	0.002
*gen*200_*p*0.9_55	4201	0.001	85426	0.004
*gen*400_*p*0.9_55	8562	0.006	232247	0.015
*gen*400_*p*0.9_65	11709	0.007	384218	0.024
*gen*400_*p*0.9_75	16388	0.009	1453981	0.084
*hamming*10-2	4	0.005	131834	0.019
*hamming*10-4	45013477	94.842	2721551	**0.247**
*hamming*6-2	4	0.000	558	0.000
*hamming*6-4	1896	0.000	1159	0.000
*hamming*8-2	4	0.000	8380	0.001
*hamming*8-4	303	0.000	27950	0.005

Tabela 3 – Continuação dos Resultados

*johnson*16-2-4	2216795	0.086	1219880	**0.039**
*johnson*32-2-4	*aborted*	*aborted*	*aborted*	*aborted*
*johnson*8-2-4	460	0.001	1105	**0.000**
*johnson*8-4-4	211	0.000	3476	0.000
*keller*4	82646	0.018	1675425	0.069
*keller*5	3623257109	1930.448	*aborted*	*aborted*
*p_hat*1000-1	254937521	71.148	1323809271	99.093
*p_hat*1000-2	364675172	171.837	8575045552	760.101
*p_hat*1000-3	558151141	350.086	11820981940	1299.626
*p_hat*300-1	283383	0.043	276845	**0.014**
*p_hat*300-2	283126	0.064	1435368	0.071
*p_hat*300-3	228584	0.075	2973494	0.154
*p_hat*500-1	3924453	0.705	21569803	1.168
*p_hat*500-2	5881265	1.543	31349229	1.886
*p_hat*500-3	6438257	2.249	107050505	7.763
*p_hat*700-1	42615280	10.364	282806924	17.056
*p_hat*700-2	35235961	14.410	262129917	20.184
*p_hat*700-3	31214587	17.919	666515473	55.069
*san*1000	10778	0.037	3114873	0.326
*san*200_0.7_1	4330	0.001	186213	0.012
*san*200_0.7_2	1939	0.001	168319	0.011
*san*200_0.9_1	1850	0.001	29456	0.002
*san*200_0.9_2	3540	0.001	27030	0.003
*san*200_0.9_3	2085	0.001	53457	0.004
*san*400_0.5_1	1315	0.002	187692	0.016
*san*400_0.7_1	16229	0.014	3517492	0.209
*san*400_0.7_2	7973	0.008	3524210	0.204
*san*400_0.7_3	10361	0.009	4144562	0.244
*san*400_0.9_1	19054	0.009	154281	0.014
*sanr*200_0.7	127080	0.025	1925452	0.086
*sanr*200_0.9	4095	0.001	44697	0.003
*sanr*400_0.5	13683397	3.322	131353597	6.833
*sanr*400_0.7	3370144	1.075	34247540	1.946

Na tabela de 2 é possível visualizar, destacados em negrito, os tempos em que o algoritmo desenvolvido foi melhor que o algoritmo de (MCCREESH; PROSSER, 2014).

Apesar de possuir um tempo de execução pior que o algoritmo de (MCCREESH; PROSSER, 2014) na maioria dos casos, existe uma instância em que o tempo é reduzido de 94 segundos para menos de 1 segundo quando é utilizada a nova solução, por isso, acredita-se que continuar trabalhando no melhoramento desta solução é viável e podem-se obter resultados ainda melhores.

Utilizar a partição em cliques para ordenar os vértices em cada subproblema é uma

opção utilizada em (MCCREESH; PROSSER, 2014) que não é utilizada na nossa solução. Isso pode impactar diretamente no tempo gasto para encontrar bons limite superiores.

Substituir o algoritmo de (MCCREESH; PROSSER, 2014) pelo nosso algoritmo ainda não parece ser uma opção viável pois são poucos casos em que obtemos uma resposta melhor, porém, acredita-se que com algumas alterações e testes é possível equiparar ou até mesmo tornar o algoritmo melhor que o estado da arte.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho contribui desenvolvendo um novo algoritmo capaz de solucionar o problema de encontrar uma biclique induzida máxima em um grafo arbitrário. A solução desenvolvida utiliza como regra principal de poda a técnica das Bonecas Russas.

É possível concluir que a utilização da técnica das Bonecas Russas é capaz de ajudar a resolver o problema de encontrar uma biclique induzida máxima.

Para o algoritmo desenvolvido, ainda existem alternativas capazes de torná-lo melhor. Como trabalhos futuros, dentre os fatores que podem ser melhorados ou testados no algoritmo, alguns são:

1. Estudar novos critérios de ordenação dos vértices;
2. Utilizar outras políticas de ordenação;
3. Adotar o algoritmo de partição em cliques para definir a ordem de ramificação para cada subproblema;
4. Utilizar algoritmos de coloração.
5. Realizar testes com instâncias diferentes.
6. Desenvolvimento de um algoritmo híbrido.
7. Desenvolver um novo algoritmo para biclique induzida desbalanceada máxima.

REFERÊNCIAS

CORRÊA, R. C.; MICHELON, P.; CUN, B. L.; MAUTOR, T.; DONNE, D. D. A bit-parallel russian dolls search for a maximum cardinality clique in a graph. **arXiv preprint arXiv:1407.1209**, 2014.

DIMACS. **DIMACS BENCHMARK**. 2016. <http://dimacs.rutgers.edu/>. "[Online; acessado 26/05/2016]".

KIVILUOTO, L.; ÖSTERGåRD, P.; VASKELAINEN, V. Algorithms for finding maximum transitive subtournaments. **Journal of Combinatorial Optimization**, Springer US, p. 1–13, 2014.

MCCREESH, C.; PROSSER, P. An exact branch and bound algorithm with symmetry breaking for the maximum balanced induced biclique problem. In: **Integration of AI and OR Techniques in Constraint Programming**. [S.l.]: Springer, 2014. p. 226–234.

SANCHEZ, M.; ALLOUCHE, D.; GIVRY, S. de; SCHIEX, T. Russian doll search with tree decomposition. In: CITESEER. **IJCAI**. [S.l.], 2009. p. 603–608.

SEGUNDO, P. S.; MATIA, F.; RODRIGUEZ-LOSADA, D.; HERNANDO, M. An improved bit parallel exact maximum clique algorithm. **Optimization Letters**, Springer, v. 7, n. 3, p. 467–479, 2013.

SHAHINPOUR, S. **Optimization-Based Network Analysis with Applications in Clustering and Data Mining**. Tese (Doutorado) — Texas A&M University, 2013.

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

TOMITA, E.; SUTANI, Y.; HIGASHI, T.; TAKAHASHI, S.; WAKATSUKI, M. A simple and faster branch-and-bound algorithm for finding a maximum clique. In: **WALCOM: Algorithms and computation**. [S.l.]: Springer, 2010. p. 191–203.

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.

VERFAILLIE, G.; LEMAÎTRE, M.; SCHIEX, T. Russian doll search for solving constraint optimization problems. In: CITESEER. **AAAI/IAAI, Vol. 1**. [S.l.], 1996. p. 181–187.

ÖSTERGåRD, P. R. A new algorithm for the maximum-weight clique problem. **Electronic Notes in Discrete Mathematics**, v. 3, n. 0, p. 153 – 156, 1999. 6th Twente Workshop on Graphs and Combinatorial Optimization.

ÖSTERGåRD, P. R. A fast algorithm for the maximum clique problem. **Discrete Applied Mathematics**, v. 120, p. 197–207, 2002. Special Issue devoted to the 6th Twente Workshop on Graphs and Combinatorial Optimization.

ÖSTERGåRD, P. R.; VASKELAINEN, V. Russian doll search for the steiner triple covering problem. **Optimization Letters**, Springer-Verlag, v. 5, n. 4, p. 631–638, 2011.

ÖSTERGåRD, P. R.; VASKELAINEN, V.; MOSIG, A. Algorithms for the combinatorial best barbeque problem. **MATCH Communications in Mathematical and in Computer Chemistry**, v. 58, p. 309–321, 2007.