UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ENOQUE ALVES DE CASTRO NETO

ALGORITMO EXATO PARA O PROBLEMA DO K-PLEX MÁXIMO

QUIXADÁ

2018

ENOQUE ALVES DE CASTRO NETO

ALGORITMO EXATO PARA O PROBLEMA DO K-PLEX MÁXIMO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Dr. Wladimir Araújo Tavares

QUIXADÁ

2018

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)

C35a Castro Neto, Enoque Alves de.
Algoritmo exato para o problema do k-plex máximo / Enoque Alves de Castro Neto. – 2018.
35 f. : il.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Ciência da Computação, Quixadá, 2018.
Orientação: Prof. Dr. Wladimir Araújo Tavares.

1. Otimização Combinatória. 2. Teoria dos Grafos. I. Título.

CDD 004

ENOQUE ALVES DE CASTRO NETO

ALGORITMO EXATO PARA O PROBLEMA DO K-PLEX MÁXIMO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em: __/__/___

BANCA EXAMINADORA

Prof. Dr. Wladimir Araújo Tavares (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Me. Francisco Erivelton Fernandes de Aragão
Universidade Federal do Ceará (UFC)

Prof. Me. Arthur Rodrigues Araruna
Universidade Federal do Ceará (UFC)

À toda minha família, por sempre acreditar e investir em mim.

AGRADECIMENTOS

Ao Doutorando em Engenharia Elétrica, Ednardo Moreira Rodrigues, e seu assistente, Alan Batista de Oliveira, aluno de graduação em Engenharia Elétrica, pela adequação do *template* utilizado neste trabalho para que o mesmo ficasse de acordo com as normas da biblioteca da Universidade Federal do Ceará (UFC).

Aos meus pais, Antonieta Alves e Antônio Alves.

À toda a turma de Ciência da Computação de 2015 por estarem sempre apoiando uns aos outros durante essa árdua caminhada da graduação.

Ao Prof. Dr. Wladimir Araújo Tavares, por toda sua paciência durante sua orientação e por todos os ensinamentos passados no grupo de estudos da maratona de programação.

Aos Computeiros do Zodíacos, Pedro Henrique e Pedro Olímpio.

Ao João Pedro e ao Alex Sandro, por sempre me apoiarem positivamente, muitas vezes em forma de xingamentos, toda vez que eu dizia que não iria conseguir me formar.

À todos que contribuíram direta e indiretamente para a minha formação.

“Amigos podem seguir caminhos diferentes, mas não deixam de ser amigos.”

(Gon Freecs)

RESUMO

O problema do *k-plex* máximo pode ser apresentado como: Dado um grafo não direcionado $G = (V, E)$, encontre $S \subseteq V$ com a maior cardinalidade 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 (SILVA; TAVARES, 2016), ou seja, cada vértice é não adjacente de no máximo k vértices. O problema do *k-plex* máximo é uma relaxação do problema da clique máxima (TRUKHANOV *et al.*, 2013). Toda clique é uma *1-plex*. Algoritmos para encontrar o *k-plex* máximo tem como uma de suas aplicações a análise de grupos coesos em redes sociais. Neste trabalho, reunimos as técnicas apresentadas por Silva e Tavares (2016) e McClosky e Hicks (2012) para criar um novo algoritmo para o problema do *k-plex* máximo. Esse algoritmo usa a técnica *branch-and-bound* e adiciona um limite superior usando a técnica de *co-k-coloração*, proposta por McClosky e Hicks (2012). Testes computacionais foram feitos com o intuito de comparar a eficiência entre o algoritmo proposto por este trabalho e o algoritmo de Silva e Tavares (2016). Foram utilizadas entradas do DIMACS e instâncias geradas aleatoriamente e o algoritmo proposto por este trabalho se saiu melhor em 40 de 75 instâncias.

Palavras-chave: *k-plex* máximo. Otimização Combinatória. Teoria dos Grafos.

ABSTRACT

The *k-plex* maximum problem can be presented as: Given an undirected graph $G = (V, E)$, find $S \subseteq V$ with the highest cardinality such that $\forall v \in S, |\Gamma(v) \cap S| \geq |S| - k$, where $\Gamma(v)$ represents the set of vertices adjacent to v (SILVA; TAVARES, 2016), ie, each vertex is non-adjacent to at most k vertices. The problem of the maximum *k-plex* is a maximum clique relaxation (TRUKHANOV *et al.*, 2013). Every clique is a *1-plex*. Algorithms to find the k-plex maximum have as one of its applications the analysis of cohesive groups in social networks. n this paper, we have assembled the techniques presented by Silva e Tavares (2016) and McClosky e Hicks (2012) to create a new algorithm for the maximal *k-plex* problem. This algorithm uses the *branch-and-bound* technique and adds an upper bound using the *co-k-coloration* technique proposed by McClosky e Hicks (2012). Computational tests were made with the purpose of comparing the efficiency between the algorithm proposed by this work and the algorithm of Silva e Tavares (2016). DIMACS inputs and randomly generated instances were used and the algorithm proposed by this work performed best in 40 of 75 instances.

Keywords: k-plex maximum. Graph Theory. Combinatorial Optimization

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BPC	BasicPlexColoring
BPCKC	BasicPlexCoKColoring

LISTA DE SÍMBOLOS

$\omega_k(G)$	Maior k-plex do grafo G
$\Gamma(v)$	Conjunto de vértices adjacente a v
$\Delta(G)$	Maior Grau do grafo G

SUMÁRIO

1 INTRODUÇÃO

Um importante campo de estudo da Matemática e Ciência da Computação é a Teoria dos Grafos. Um grafo representa as relações entre objetos de determinado conjunto, onde esses objetos são chamados de vértices e as relações entre eles são chamadas de arestas. Vários problemas reais podem ser modelados em forma de grafos, como por exemplo: saber a menor distância possível entre duas cidades, escolher a melhor rota a ser seguida por um entregador, mapeamento de relacionamento entre pessoas em redes sociais, entre outros.

Um grafo G pode ser representado por $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas. Uma aresta $e \in E$ é representada pela a relação $\{\{x, y\} \mid x, y \in V \ e \ x \neq y\}$. Quando o conjunto de aresta possui todas as arestas possíveis do grafo, este grafo é chamado de grafo completo. Encontrar o maior subconjunto de vértice de um grafo, tal que o novo grafo gerado seja um grafo completo, é um problema muito conhecido na teoria dos grafos e é chamado de problema da clique máxima. O problema da clique máxima é um problema NP-Difícil clássico de fundamental importância para a otimização combinatória. O problema da clique máxima possui diversas aplicações, como: Bioquímica (BUTENKO; WILHELM, 2006), mineração de dados (WASHIO; MOTODA, 2003), bioinformática e processamento de imagens (TOMITA; SEKI, 2003).

O problema do *k-plex* máximo é uma relaxação do problema da clique máxima (TRUKHANOV *et al.*, 2013). Ele pode ser definido como: $\forall v \in S, |\Gamma(v) \cap S| \geq |S| - k$, em que $\Gamma(v)$ representa o conjunto de vértices adjacentes de v (SILVA; TAVARES, 2016), ou seja, cada vértice é não-adjacente de no máximo k vértices. Uma clique é uma 1-plex, pois em uma clique todos os vértices da clique são não-adjacentes a exatamente um vértice, ele mesmo (SILVA; TAVARES, 2016).

O *k-plex* máximo em um grafo tem como algumas de suas aplicações a analise de coesão social em uma determinada rede social e com isso desenvolver teorias sociológicas (BALASUNDARAM *et al.*, 2011). Os principais algoritmos existentes para esse problema foram propostos em McClosky e Hicks (2012) e Trukhanov *et al.* (2013).

O problema da clique máxima e o problema do *k-plex* são problemas NP-Difíceis, sendo assim, nenhum algoritmo eficiente é conhecido para resolve-los. Os métodos que são comumente usados para resolver esses problemas são os métodos de otimização combinatória e o método de programação linear inteira (PLI). Tanto a otimização combinatória quanto a programação linear inteira geralmente utilizam um algoritmo *branch-and-bound*, que consiste

em testar todas as possíveis possibilidades usando algumas restrições previamente conhecidas. Existem várias formas de definir as restrições de um algoritmo *branch-and-bound*. A eficiência desses algoritmos geralmente dependem do quão bem foi definido suas restrições. Para o problema da clique máxima e *k-plex* máxima, um exemplo de restrição é um limite superior possível como solução.

Silva e Tavares (2016) usaram um algoritmo baseado na coloração em grafos para calcular um limite superior para todos os vértices do conjunto candidato, eliminando assim os vértices que não podem melhorar a solução. Também estabelece uma ordem fixa para adicionar os vértices a uma possível solução. Já McClosky e Hicks (2012) utilizam como limite superior a co-k-coloração de vértices, mas não utilizam uma ordem fixa de limite superior, precisando recalcular esse limite para todo subproblema. Neste trabalho foi utilizado a co-k-coloração, proposta por McClosky e Hicks (2012), junto com a ordem fixa de vértice, proposta por Silva e Tavares (2016). Também foi realizado os testes computacionais, comparando o algoritmo proposto com o algoritmo de Silva e Tavares (2016).

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Problema do *k*-plex máximo

Dado um grafo $G = (V,E)$, onde V é o conjunto de vértices e E o conjunto de arestas. Uma clique, em grafos, é um grafo tal que esse grafo é completo. A cardinalidade de uma clique é igual ao numero de vértices pertencente ao grafo, por exemplo, uma clique de cardinalidade 3 possui 3 vértices. Um problema muito conhecido na Ciência da Computação é o problema da clique máxima em um grafo, que consiste em achar a clique de maior cardinalidade em um subgrafo induzido. Esse problema é um problema NP-Difícil clássico de fundamental importância para a otimização combinatória.

O problema do k-plex máximo é uma relaxação do problema da clique máxima (TRUKHANOV *et al.*, 2013). Ele pode ser definido como: Dado um grafo $G = (V,E)$, um k-plex é um subconjunto $S \subseteq V$ 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 (SILVA; TAVARES, 2016), ou seja, cada vértice é não adjacente de no máximo k vértices. Uma clique é uma 1-plex, pois em uma clique todos os vértices da clique são não adjacentes a exatamente 1 vértice (ele mesmo).

2.2 Co-k-plex

Em um grafo $G = (V,E)$, um *co-k-plex* é um subconjunto $C \subseteq V$ tal que $\Delta[C] \leq k-1$, onde $\Delta[C]$ é o grau máximo do conjunto de vértice C (SEIDMAN; FOSTER, 1978). Se um conjunto C é *co-k-plex* em G, então ele também é *k-plex* em $\overline{G}$ (MCCLOSKY; HICKS, 2012).

2.3 Problema da co-k-coloração mínima

Dado um grafo $G = (V,E)$, uma cor é definida como um conjunto $C \subseteq V$ tal que cada vértices pertencente a C tenha no máximo $k-1$ vértices adjacentes também pertencente a C. O problema da *co-k-colorão* miníma consiste no menor número de cor possíveis para um determinado grafo. No final, essa coloração gerará vários *co-k-plex*'s.

McClosky e Hicks (2012) utiliza essa técnica para conseguir o limite superior para o problema da clique máxima. Neste trabalho essa técnica é utilizada para conseguir o limite superior para o problema do *k-plex* máximo.

2.4 Limite superior para o algoritmo *branch-and-bound*

Dado um *co-k-plex* $C \subseteq G$, McClosky e Hicks (2012) definem e demostram 4 possíveis limites superiores, que são eles:

1. O tamanho do conjunto C, dado por $|C|$.
2. $2k-2+(k \ mod \ 2)$, em que k é o inteiro que representa o tipo do *co-k-plex*.
3. $\Delta(G[C])+k$, em que $\Delta(G[C])$ é o grau máximo de C e k é o inteiro que representa o tipo do *co-k-plex*.
4. $k+J$, em que $J \leftarrow max \ \{m : a_m \geq k+m\}$ e a_m é o numero de vértices com grau maior ou igual a m.

O mínimo entre esses quatro limites apresentados será o limite superior usado no algoritmo *branch-and-bound*.

2.5 Algoritmo *branch-and-bound*

A técnica ramificar-e-podar (*branch-and-bound*) é uma importante ferramenta para a produção de um algoritmo de solução exata para um problema NP-Difícil (MORRISON *et al.*, 2016). A técnica consiste em gerar vários subproblemas e abandonar os não promissores. Para abandonar um subproblemas, definimos um procedimento de limite superior, assim que for descoberto que um subproblema não consegue superar a melhor solução atual, ele se torna infrutífero e pode ser abandonado.

3 TRABALHOS RELACIONADOS

McClosky e Hicks (2012) apresentam um algoritmo combinatório para a resolução do problema da clique máxima e uma adaptação deste algoritmo para resolver o problema do *k-plex* máximo. Os autores utilizam a co-k-coloração para encontrar o limite superior para o algoritmo de *branch-and-bound*, porém seu algoritmo não possui uma ordem fixa na sua utilização, ou seja, uma ordem fixa de vértices a serem testados pelo o algoritmo, por isso, é necessário recalcular a co-k-coloração para todo novo subproblema gerado pelo o algoritmo.

Silva e Tavares (2016) propõem um algoritmo para o problema do *k-plex* máximo baseado na heurística de coloração, lista de vértices saturados, ordem fixa e paralelismo de bits. É usado como limite superior a coloração de grafo. O algoritmo de Silva e Tavares (2016) é usado como base pare este trabalho, porém o limite superior do algoritmo é adquirido a partir do resultado da co-k-coloração e não será usado a técnica de paralelismo de bits.

Trukhanov *et al.* (2013) propõem uma solução para o problema da clique máxima e o *k-plex* máximo utilizando um algoritmo genérico que utiliza a técnica *Russian Doll Search* (RDS). Essa técnica consiste em resolver um conjunto de subproblemas iterativamente e armazenar seu valor ótimo. Caso esse subproblema apareça novamente durante o processo de *branch-and-bound*, utiliza-se o resultado previamente calculado como polda.

Neste trabalho, é desenvolvido um algoritmo *branch-and-bound* baseado no algoritmo de Silva e Tavares (2016), porém utilizando o limite superior proposto por McClosky e Hicks (2012) para o problema da clique máxima. Também será usada uma ordem fixa de limite superior por vértice para o limite superior, assim não precisando recalcular esse limite para subproblema gerado pelo algoritmo.

4 PROCEDIMENTOS METODOLÓGICOS

4.1 Algoritmo base

Um subproblema do problema do *k-plex* máximo pode ser definido como uma tupla (S, U, S_{max}), onde S é o *k-plex* que está sendo construído, U é o conjunto de candidatos formado pelos vértices que podem entrar no *k-plex* que está sendo construído, e o S_{max} é a solução de maior cardinalidade encontrada até o momento (SILVA; TAVARES, 2016).

Um subproblema (S, U, S_{max}) é podado, quando $|S| + |U| \leq S_{max}$, ou seja, não importa o quanto o algoritmo execute deste ponto para frente, ele não melhorará o resultado, então o subproblema pode ser abandonado. Caso isso não aconteça, um vértice $v \in U$ é escolhido para entrar em S e o conjunto candidato deve ser atualizado. O subproblema obtido será $(S \cup \{v\}, U', S_{max})$, sendo que

$$U' = \{u \in U \setminus \{v\} | S \cup \{v\} \cup \{u\} \text{é } k\text{-plex}\}$$

Silva e Tavares (2016) apresentam uma função chamada de *BasicPlex* que está presente na Figura 1. Esse Algoritmo é o procedimento base desenvolvido neste trabalho. O algoritmo apresentado na Figura 1 é um procedimento *branch-and-bound*, ou seja, adota um procedimento de poda para eliminar ramificações infrutíferas que correspondem a ramificações que não darão a melhor resposta.

A entrada do *BasicPlex* é um subproblema do *k-plex*, isto é, uma tupla (S, U, S_{max}), onde S é o *k-plex* que está sendo construído, U é o conjunto candidato formado pelos os vértices que podem entrar no *k-plex* que está sendo construído, e o S_{max} é a solução de maior cardinalidade encontrada até o momento. Inicialmente o conjunto S e o conjunto S_{max} são vazios e o conjunto U possui todos os vértices, pois todos são candidatos a formarem um *k-plex*.

Figura 1 – Algoritmo BasicPlex

Algoritmo 1 – BasicPlex

```
1:  função BASICPLEX(S, U, k, S_max)
2:      se |S| > |S_max| então
3:          S_max ← S
4:      fim se
5:      enquanto U ≠ ∅ faça
6:          se |U| + |S| ≤ |S_max| então
7:              devolve
8:          fim se
9:          U ← U − {v} para algum v ∈ U
10:         S ← S ∪ {v}
11:         U' ← {u ∈ U : S ∪ {u} é um k-plex }        // O(|U| × |S|^2)
12:         BasicPlex (S, U', k, S_max)
13:         S ← S − {v}
14:     fim enquanto
15: fim função
```

Figura 2 – Fonte: Silva e Tavares (2016)

4.2 Gerando o conjunto candidato

Quando um novo vértice é adicionado no conjunto solução, é necessário atualizar o conjunto candidato para essa nova solução. Na Figura 1, essa atualização é feita na linha 11 e tem um custo na ordem de $O(|U| \times |S|^2)$, já que para cada vértice $v \in U$ é necessário testar se $S \cup \{v\}$ é um *k-plex*. Essa operação obteve uma complexidade reduzida usando uma lista de vértices saturados (*Saturated Lists*), apresentada por Trukhanov *et al.* (2013).

Trukhanov *et al.* (2013) utiliza um algoritmo chamado *MakeSaturatedList* para a construção da lista de vértices saturados, ou seja, vértices que pertencem à solução atual S que já possuem exatamente $k-1$ não-vizinhos em S. Com a lista de vértices saturados, a atualização do conjunto candidato consiste em manter todos os vértice de U que são vizinho a todos os vértices saturados da solução atual S.

O pseudo-código da função *MakeSaturatedList* está disponível na Figura 3 e tem como entrada o conjunto candidato U, um vetor *nncnt* tal que $nncnt[v]$ representa a quantidade de vértices que estão no conjunto solução S e não são adjacentes a v. A ideia desse algoritmo consiste em: Atribuir u ao o último vértice da solução S, e atualizar a quantidade do vetor *nncnt* para todos os não adjacentes de u e adicionando ao conjunto SL para todos os vértices que

possuem exatamente $k-1$ não-adjacentes aos vértices da solução. Essa função é da ordem de $O(|U|+|S|)$.

Figura 3 – MakeSaturatedList

Algoritmo 2 MakeSaturatedList

```
1:  função MAKESATURATEDLIST(U, nncnt, S, k)
2:      SL ← ∅
3:      u ← último vértice de S
4:      para todo v ∈ S − {u} faça
5:          se (u, v) ∉ G então
6:              nncnt[v] ← nncnt[v] + 1
7:          fim se
8:      fim para todo
9:      para todo v ∈ U faça
10:         se (u, v) ∉ G então
11:             nncnt[v] ← nncnt[v] + 1
12:         fim se
13:     fim para todo
14:     para todo v ∈ S faça
15:         se nncnt[v] = k − 1 então
16:             SL ← SL ∪ {v}
17:         fim se
18:     fim para todo
19:     devolve SL
20: fim função
```

Fonte: Trukhanov *et al.* (2013) com adaptações de Silva e Tavares (2016)

Depois da construção da lista de vértices saturados, para saber se um vértice pode entrar no conjunto candidato basta percorrer todo o conjunto de vértice saturado e testar se esse vértice é vizinho de todos os vértices saturados, caso contrário a entrada desse vértice faria o conjunto S deixar de ser *k-plex* e com isso ele não é um candidato para essa solução. Uma função chamada *IsPlex* é criada com essa propriedade e seu pseudo-código está descrito na Figura 4. Sua complexidade é da ordem de $O(|SL|)$.

Figura 4 – IsPlex

```
1: função ISPLEX(SL, v)
2:     para todo u ∈ SL faça
3:         se (u, v) ∉ G então
4:             devolve falso
5:         fim se
6:     fim para todo
7:     devolve verdade
8: fim função
```

Fonte: Trukhanov *et al.* (2013) com adaptações de Silva e Tavares (2016)

A função *Generation*, que tem seu pseudo-código descrito na figura 5 recebe como entrada o conjunto candidato U, o conjunto solução S, o vetor *nncnt* e um k que representa o tipo do *k-plex*. Essa função uni a função *MakeSaturatedList* e *IsPLex* para gerar o novo conjunto de candidatos. A complexidade da função *Generation* é na ordem $O(|U| \times |SL| + |S| + |U|)$.

Figura 5 – Generation

```
1: função GENERATION(U, nncnt, S, k)
2:     R ← ∅
3:     SL ← MakeSaturatedList(U, nncnt, S, k)
4:     para todo v ∈ U faça
5:         se nncnt[v] > k − 1 então
6:             continue
7:         fim se
8:         se isPles(SL, v) então
9:             R ← R ∪ {c}
10:        fim se
11:    fim para todo
12:    devolve (R, nncnt)
13: fim função
```

Fonte: Silva e Tavares (2016)

Ajustando a função *BasicPlex* para gerar o novo conjunto candidato usando a função *Generation* fica como demostrado na figura 6

Figura 6 – BasicPlex (Com a função Generation)

```
1: função BASICPLEX(S,U,K,S_max,nncnt)
2:     se |S| > |S_max| então
3:         S_max ← S
4:     fim se
5:     enquanto U ≠ ∅ faça
6:         se |U| + |S| ≤ |S_max| então
7:             devolve
8:         fim se
9:         U ← U − {v} para algum v ∈ U
10:        S ← S ∪ {v}
11:        (U', newNncnt) ← Generation(U,K,S,nncnt)        // O(|U| × |SL| + |S| + |U|)
12:        BasicPlex (S,U',k,S_max,newNncnt)
13:        S ← S − {v}
14:    fim enquanto
15: fim função
```

Fonte: McClosky e Hicks (2012) com adaptações de Silva e Tavares (2016)

4.3 Adicionando o limite superior obtido da co-k-coloração

Dado um *co-k-plex* $C \subseteq G$, McClosky e Hicks (2012) definem e provam 4 possíveis limites superiores, que são eles:

1. O tamanho do conjunto C, dado por $|C|$.
2. $2k-2+(k \ mod \ 2)$, em que k é o inteiro que representa o tipo do *co-k-plex*.
3. $\Delta(G[C])+k$, em que $\Delta(G[C])$ é o grau máximo de C e k é o inteiro que representa o tipo do *co-k-plex*.
4. $k+J$, em que $J \leftarrow max\ \{m : a_m \geq k+m\}$ e a_m é o numero de vértices com grau maior ou igual a m.

Em uma *co-k-coloração*, cada cor gerada pelo algoritmo é um *co-k-plex* válido. Com isso, o limite superior do grafo será calculado pelos co-k-plex gerados pela *co-k-coloração*. O algoritmo da *co-k-coloração* está descrito na figura 7.

A *co-k-coloração* descrito na figura 7 recebe um grafo G e um K que representa o tipo do *k-plex*, e retorna o limite superior sequencial dos vértices G. É usado as funções *pode_entrar_na_cor* e *coloca_vertice_na_cor* que são descrito respectivamente na figura 7 e 9.

Figura 7 – co-k-coloração

```
1: função CO-K-COLORAÇÃO(G, K)
2:     cores.push_back(∅)    //Adiciona uma nova cor ao conjunto de cores
3:     vizinhos ← vetor de tamanho da quantidade de vertice
4:     flag ← verdade
5:     para todo v ∈ G faça
6:         para todo cor ∈ cores faça
7:             se (pode_entrar_na_cor(cores, cor, v, G, vizinhos, K)) então
8:                 coloca_vertice_na_cor(cores, cor, v, G, vizinhos)
9:                 flag ← falso
10:            fim se
11:        fim para todo
12:        se (flag) então
13:            cores.push_back(∅)
14:            nova_cor ← |cores|
15:            coloca_vertice_na_cor(cores, nova_cor, v, G, vizinhos)
16:        fim se
17:    fim para todo
18:    devolve cores
19: fim função
```

Fonte: Autor

Figura 8 – pode_entrar_na_cor

```
1: função PODE_ENTRAR_NA_COR(cores, cor, v, G, vizinhos, K)
2:     para todo c ∈ cores[cor] faça
3:         se (c, v) ∈ G então
4:             se vizinhos[c] ≥ K − 1 então
5:                 devolve falso
6:             fim se
7:         fim se
8:     fim para todo
9:     devolve verdade
10: fim função
```

Fonte: Autor

Figura 9 – coloca_vertice_na_cor

```
1: função COLOCA_VERTICE_NA_COR(cores, cor, v)
2:     contador ← 0
3:     para todo c ∈ cores[cor] faça
4:         se (c, v) ∈ G então
5:             vizinhos[c] ← vizinhos[c] + 1
6:             contador ← contador + 1
7:         fim se
8:     fim para todo
9:     cores[cor] ← cores[cor] ∪ {v}
10:    vizinhos[v] ← contador
11: fim função
```

Fonte: Autor

Para gerar o limite superior a partir do co-k-coloração, será usada a função *gera_limite* descrita na figura 10. A sua entrada é um grafo G e um K que representa o tipo do *k-plex* e retorna uma tupla contendo o conjunto contendo a ordem de vértice a ser seguida e o limite superior para esse vértice. Sua ordem de complexidade é $O(|V|^3)$

Figura 10 – gera_limite

```
1: função GERA_LIMITE(G, K)
2:     cores ← co_k_coloracao(G, K)          //O(|V[G]|^3)
3:     ordem_vertices ← ∅
4:     limite_superior ← ∅
5:     para todo cor ∈ cores faça
6:         J_cor ← max{m : a_m ≥ K + m}    para Cores[cor]
7:     fim para todo
8:     bound ← 0
9:     para todo cor ∈ cores faça
10:        para todo v ∈ cores[cor] faça
11:            menor ← min{2K − 2 + K mod 2, J_cor, Δ(G[cores[cor]]) + k, |cores[cor]|}
12:            limite_superior ← limite_superior ∪ {bound + menor}
13:            ordem_vertices ← ordem_vertices ∪ {v}
14:        fim para todo
15:        bound ← bound + min{2K − 2 + K mod 2, J_cor, Δ(G[cores[cor]]) + k, |cores[cor]|}
16:    fim para todo
17:    devolve (ordem_vertices, limite_superior)
18: fim função
```

Fonte: Autor

A refatoração do código do *BasicPlex* para adicionar o limite obtido pela a co-k-coloração é descrito na figura 10. A função *gera_limite* retorna a ordem dos vértice e seus respectivos limites superiores em ordem crescente. Na linha 9, é testado se o limite superior do vértice atual mais o tamanho da solução atual é capaz de ser maior que a maior solução encontrada até o momento. Caso contrário, já é possível realizar a poda nesse ramo, pois os vértices são testados em ordem decrescente, garantindo assim que todos os outros vértices restantes também não satisfará a condição.

Figura 11 – BasicPlex (Com co-k-coloração)

```
1:  função BASICPLEX(S, U, k, S_max, nncnt)
2:      se |S| > |S_max| então
3:          S_max ← S
4:      fim se
5:      (limites, ordem) ← gera_limite(U, K)
6:      enquanto ordem ≠ ∅ faça
7:          v ← ultimo vertice de ordem
8:          limite ← ultimo elemento do conjunto de limites
9:          se limite + |S| ≤ |S_max| então
10:             devolve
11:         fim se
12:         U ← U − {v}
13:         S ← S ∪ {v}
14:         ordem ← ordem − {v}
15:         limites ← limites − {limite}
16:         (U', newNncnt) ← Generation(U, k, S, nncnt)        // O(|U| × |SL| + |S| + |U|)
17:         BasicPlex (S, U', k, S_max, newNncnt)
18:         S ← S − {v}
19:     fim enquanto
20: fim função
```

Fonte: McClosky e Hicks (2012) com adaptções do Autor.

5 RESULTADOS

Os testes foram realizados em um computador com 8 GB de memória RAM, processador Intel core i5-3470 CPU @ 3.2XGHz x 4 e sistema operacional Linux.

Em Silva e Tavares (2016), o algoritmo **BPC** foi implementado usando a técnica de paralelismo de bits. Já para os testes computacionais deste trabalho, o algoritmo foi implementado sem a técnica de paralelismo de bits.

Para a execução dos testes foram utilizadas entradas do DIMACS e instâncias geradas aleatoriamente. As descrições das entradas estão no Apêndice A.

A tabela Tabela 1 apresenta a comparação entre o algoritmo **BasicPlexCoKColoring (BPCKC)** e o algoritmo **BasicPlexColoring (BPC)** proposto por Silva e Tavares (2016). Foi considerado como parâmetro de comparação o tamanho do maior *k-plex* ($\omega_2(G)$) encontrado pelo o algoritmo e como critério de desempate o tempo gasto para encontrar este *k-plex*.

Todos os testes tiveram um tempo limite de 3600s e foi considerado $k = 2$. Os valores destacados indicam o algoritmo com melhor desempenho naquela instância.

Tabela 1 – Resultados dos algoritmos de *BPCKC* e *BPC*

Grafo	BPC		BPCKC	
	$\omega_2(G)$	segundos	$\omega_2(G)$	segundos
brock200_1	23	2753.30	**24**	**3146.82**
brock200_2	**13***	**2.41**	13*	16.31
brock200_3	**17***	**37.11**	17*	248.88
brock200_4	**19**	**798.66**	19	801.32
brock400_1	**26**	**2882.28**	26	2996.44
brock400_2	**26**	**268.34**	26	1531.41
brock400_3	25	1190.76	**26**	**1705.52**
brock400_4	26	1797.14	**26**	**92.74**
brock800_1	21	48.11	**22**	**302.76**
brock800_2	21	63.53	**22**	**485.18**
brock800_3	22	127.71	**23**	**1345.95**
brock800_4	21	1086.55	**22**	**760.62**
C2000.5	17	139.33	**18**	**3055.90**

Continua na próxima página...

Grafo	BPC		BPCKC	
	$\omega_2(G)$	segundos	$\omega_2(G)$	segundos
C2000.9	64	432.76	**65**	**2552.92**
C4000.5	**18**	**6.98**	18	**68.60**
c-fat200-1	12*	0.1	12*	0.1
c-fat200-2	23	0.1	23	0.1
c-fat200-5	58	0.1	58	0.1
c-fat500-1	14*	0.1	14*	0.1
c-fat500-2	26*	0.1	26*	0.1
c-fat500-5	64*	0.1	64*	0.1
c-fat500-10	126	0.1	126	0.1
DSJC500_5	15	387.87	**15**	**91.60**
DSJC1000_5	16	34.79	**17**	**1622.99**
gen200_p0.9_44	**41**	**1222.58**	41	1484.10
gen200_p0.9_55	43	1446.85	**44**	**668.61**
gen400_p0.9_55	**52**	**162.76**	52	347.76
gen400_p0.9_65	48	972.74	**49**	**770.02**
gen400_p0.9_75	49	1243.01	**52**	**1724.23**
hamming6-2	32*	0.001	32*	0.001
hamming6-4	6*	0.1	6*	0.1
hamming8-2	128	0.1	128	0.1
hamming8-4	16	0.1	16	0.1
hamming10-2	**512**	**2.3**	512	2.6
hamming10-4	32	0.213	**35**	**1476.74**
johnson8-2-4	5*	0.1	5*	0.1
johnson8-4-4	14*	0.1	14*	0.1
johnson16-2-4	10	0.1	10	0.1
johnson32-2-4	21	0.1	21	0.1
keller4	15*	63.53	**15***	**7.16**
keller5	26	207.33	**27**	**144.98**
keller6	42	916.96	**52**	**490.87**

Continua na próxima página...

Grafo	BPC		BPCKC	
	$\omega_2(G)$	segundos	$\omega_2(G)$	segundos
MANN_a9	26*	0.1	26*	0.1
MANN_a27	236	285.05	**236**	**0.648**
MANN_a45	**661**	**6.4**	661	7.4
p_hat300-1	10*	0.1	10*	0.1
p_hat300-2	26	1149.72	**28**	**258.753**
p_hat300-3	38	719.41	**39**	**3262.329**
p_hat500-1	12*	21.491	**12***	**6.333**
p_hat500-2	33	824.557	**36**	**73.124**
p_hat500-3	40	2590.482	**47**	**3484.411**
p_hat700-1	13*	816.845	**13***	**98.231**
p_hat700-2	32	2548.603	**37**	**925.717**
p_hat700-3	41	383.757	**49**	**60.781**
p_hat1000-1	13*	1067.371	**13***	**53.106**
p_hat1000-2	35	35.158	**44**	**1708.827**
p_hat1000-3	45	0.197	**54**	**1145.408**
p_hat1500-1	13	79.273	14	**1104.837**
p_hat1500-2	41	5.989	**49**	**267.027**
p_hat1500-3	57	586.895	**69**	**508.671**
san200_0.7_1	29	0.021	29	0.021
san200_0.7_2	24	0.01	24	0.01
san200_0.9_1	45	644.17	**67**	**754.05**
san200_0.9_2	49	58.71	**49**	**5.40**
san200_0.9_3	40	1145.75	**41**	**2793.43**
san400_0.5_1	14	0.1	14	0.1
san400_0.7_1	**40**	**746.7**	40	1219.3
san400_0.7_2	30	0.1	30	0.1
san400_0.7_3	22	1038.0	**24**	**0.1**
san400_0.9_1	100	0.1	100	0.1
san1000	16	0.1	16	0.1

Continua na próxima página...

Grafo	BPC		BPCKC	
	$\omega_2(G)$	segundos	$\omega_2(G)$	segundos
sanr200_0.7	21	987.4	**22**	**1886.5**
sanr200_0.9	41	96.1	**42**	**1947.1**
sanr400_0.5	15	348.6	**15**	**246.8**
sanr400_0.7	**25**	**1109.33**	22	20.1

*solução ótima

O algoritmo **BPCKC** foi melhor em 40 instâncias. Em algumas instâncias como em *san200_0.9_1* o algoritmo **BPCKC** foi bastante superior ao **BPC**, se mostrando assim mais eficiente. Já o algoritmo **BPC** foi melhor em 10 instâncias e em 23 instâncias houve um empate. Totalizando assim 75 instâncias testadas.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho reuni as técnicas apresentadas por Silva e Tavares (2016) e McClosky e Hicks (2012) para propor um novo algoritmo para o problema do k-plex máximo. Em 75 instâncias computacionais, o algoritmo proposto por este trabalho obteve melhores resultados em 40 e empatando em 26 em relação ao algoritmo de Silva e Tavares (2016).

O algoritmo **BPCKC** propoem uma ordem fixa na hora de remover os vértices do conjunto candidato. A forma que está ordem é formada pode ser que influencie na eficiência deste algoritmo. Por questões de tempo, não foi possível explorar muito em relação a isso. Como também não foi possível realizar mais testes computacionais para $K > 2$. Estes serão apresentado a seguir como trabalhos futuros:

a) Adicionar a técnica de paralelismo de bits.
b) Realizar comparações entre **BPCKC** e **BPC** com $K > 2$.
c) Propor uma solução onde a ordem dos vértices do conjunto candidato seja gerado aleatoriamente para assim tentar fugir do pior caso e apostar no caso médio.

Os trabalhos usados como referência foram de extrema importância para a realização deste trabalho. Principalmente os trabalhos de Silva e Tavares (2016) e McClosky e Hicks (2012) que serviram como base para todo este trabalho.

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.

BUTENKO, S.; WILHELM, W. E. Clique-detection models in computational biochemistry and genomics. **European Journal of Operational Research**, Elsevier, v. 173, n. 1, p. 1–17, 2006.

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.

MORRISON, D. R.; JACOBSON, S. H.; SAUPPE, J. J.; SEWELL, E. C. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. **Discrete Optimization**, Elsevier, v. 19, p. 79–102, 2016.

SEIDMAN, S. B.; FOSTER, B. L. A graph-theoretic generalization of the clique concept. **Journal of Mathematical sociology**, Taylor & Francis, v. 6, n. 1, p. 139–154, 1978.

SILVA, M. R. C. da; TAVARES, W. A. **Algoritmos para o problema do k-plex máximo utilizando paralelismo de bits e coloração**. 2016.

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.

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.

WASHIO, T.; MOTODA, H. State of the art of graph-based data mining. **Acm Sigkdd Explorations Newsletter**, Acm, v. 5, n. 1, p. 59–68, 2003.

APÊNDICE A – INSTÂNCIAS

Tabela 2 – 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
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

Continua na próxima página...

Grafo	$\|V\|$	$\|E\|$	Densidade
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
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

Continua na próxima página...

Grafo	$\|V\|$	$\|E\|$	Densidade
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