Algoritmos para compor uma ferramenta web: geração de grafos grades e cordais e solução de conjunto independente, clique e emparelhamento em grafos

Daniel Campos da Silva[1] (dcamposfsa@gmail.com)
Hebert Coelho da Silva[1] (hebert@ufg.br)
[1]Instituto de Informática - Universidade Federal de Goiás

Abstract. *In this work we introduce, describe and implement algorithms for generation of graphs and solving classical problems, aggregating to the web platform Graph Problems, aiming to didactical and scientific use. We approach the generation of chordal graphs, grids and cartesian products, as well as solving the maximum clique, maximum independent set and maximum matching problems.*

Resumo. *Neste trabalho, introduzimos, descrevemos e implementamos algoritmos de geração de grafos e solução de problemas clássicos, agregando à plataforma web Graph Problems, para fins didáticos e científicos. Abordamos a geração de grafos cordais, grades e produtos cartesianos, bem como a solução dos problemas da clique máxima, conjunto independente máximo e emparelhamento máximo.*

1. Introdução

A teoria dos grafos é um campo de estudos essencial para a ciência da computação, tratando de uma das mais abrangentes estruturas de dados que são os grafos. Este campo é utilizado no dia a dia de profissionais de sistemas de informação e análise de dados, de pesquisadores acadêmicos ou empresariais, bem como professores e estudantes da área da informática.

Dois ramos interessantes da teoria dos grafos são: dividir grafos em famílias que compartilham características comuns e outro é a modelagem de problemas que podem ser resolvidos por algoritmos em grafos. Uma forma acessível de auxiliar esse processo é através de uma ferramenta *web* que tanto permita a interação com os grafos em si, quanto também a execução de algoritmos neles. Entretanto, soluções como em [UnickSoft 2022] e [Halim 2015], embora visualmente satisfatórias, não abordam uma ampla gama de algoritmos imprescindíveis para a teoria dos grafos, nem possuem opções para geração de famílias específicas. Visando apoiar a geração de grafos e execução de seus algoritmos, expandimos a ferramenta *web Graph Problems* desenvolvida em [da Silva 2018] agregando a geração de grafos cordais, e produtos cartesianos, além de algoritmos para a solução dos problemas da clique máxima, conjunto independente máximo e emparelhamento máximo.

O trabalho está organizado da seguinte forma: na Seção 2 revisamos a teoria necessária para entender os algoritmos desenvolvidos; na Seção 3 descrevemos algoritmos para gerar grafos cordais, grades e produtos cartesianos; na Seção 4 abordamos os problemas da clique, conjunto independente e emparelhamento máximos; por fim, na Seção 5 apresentamos algumas considerações finais sobre o trabalho.

2. Conceitos Básicos

Usaremos as notações e definições padrões em teoria dos grafos presentes no livro [Bondy and Murty 2008]. Um *grafo* G é um par ordenado $(V(G), E(G))$, onde $V(G)$ é o conjunto de vértices e $E(G)$ é o conjunto de arestas que ligam pares de $V(G)$. Chamamos $|V(G)| = n$ e $|E(G)| = m$ de *ordem* e *tamanho* de G, respectivamente.

Dizemos que uma aresta $\{v_i, v_j\}$ é *incidente* aos vértices v_i e v_j, e vice-versa. Chamamos de *adjacentes* tanto um par de vértices incidentes a uma aresta em comum, quanto um par de arestas incidentes a um vértice em comum. A *vizinhança*, também chamada de *vizinhança aberta*, de um vértice v é o conjunto de todos os vértices adjacentes a v, denotado por $N_G(v)$, ou apenas $N(v)$. Para uma aresta $\{v_i, v_j\}$, nomeamos-a por *laço* se $v_i = v_j$, ou simplesmente *aresta* se $v_i \neq v_j$. Quando duas ou mais arestas incidem sobre o mesmo par de vértices as chamamos de *paralelas*. Se não houver laços ou arestas paralelas em G, se trata de um *grafo simples*. O *grau* $d(v)$ de um vértice $v \in V$ é o número de arestas incidentes a v.

Chamamos H de *subgrafo* de G se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Se H for um subgrafo de G obtido a partir da exclusão de um conjunto X de vértices em G, juntamente com as arestas incidentes a X, denotamos-o por $H = G \setminus X$ e dizemos que é um subgrafo *induzido*. Existem várias formas de representar grafos, portanto convencionamos neste trabalho a utilização de círculos para vértices e linhas para arestas, como na Figura 1.

Neste trabalho, quando não especificado, considere que o grafo é simples, não nulo e seus conjuntos de arestas e vértices são finitos. Também simplificaremos a notação em alguns momentos omitindo G para $V(G)$ e $E(G)$, chamando-os, respectivamente, de V e E.

Os modelos mais comuns para representar grafos computacional e matematicamente são: *matriz de adjacência* (A_G) de dimensão $n \times n$ com elementos a_{uv} representando quantas vezes o vértice u incide sobre v; *matriz de incidência* (M_G) de dimensão $n \times m$ com elementos m_{ve}, representando quantas vezes a aresta e incide sobre v; e *lista de adjacências*, listando ordenadamente $N(v)$ para cada vértice v.

O *complemento* de um grafo G é $\bar{G}$, com o mesmo conjunto de vértices e $\{v_i, v_j\} \in E(\bar{G}) \iff \{v_i, v_j\} \notin E(G)$. Definimos como $G \cup H = (V(G) \cup V(H), E(G) \cup E(H))$ a *união de grafos* e, se G e H são disjuntos, indicamos por $G + H$ a *união disjunta*.

2.1. Famílias Especiais de Grafos

Famílias agrupam grafos que respeitam regras em comum. A família dos grafos *completos* K_n é composta por grafos onde todos os n vértices são adjacentes entre si. *Grafos bipartidos* são denotados por $G[A, B]$, pois podem ser separados em uma *bipartição* $(A, B) \in V \times V$ das chamadas *partes* A e B, tal que $A \cup B = V$, $A \cap B = \emptyset$ e toda aresta em E é incidente a $v_i \in A$ e $v_j \in B$. Caso todo vértice de uma parte for adjacente a todo vértice da outra, $G[A, B]$ é denominado *grafo bipartido completo*.

Um grafo é *conexo* se em todas suas possíveis partições (A, B) com $A \neq \emptyset$ e $B \neq \emptyset$ há uma aresta $\{a, b\}$ com $a \in A$ e $b \in B$, senão é *desconexo*. Todo G desconexo pode ser dividido em $c(G)$ *componentes*, grafos conexos que não são subgrafos entre si, de forma que $G = H_1 + \ldots + H_{c(G)}$. Um *caminho* P é um grafo cujos vértices podem ser sequenciados linearmente de forma que $(v_i, v_j) \in E \iff j = i + 1$ ou $i = j + 1$. Se

P não possui nenhum vértice repetido em sua sequência, dizemos que é um *caminho simples*. Um *ciclo* C é um grafo com três ou mais vértices que podem ser ordenados numa sequência cíclica em que todo par de vértices adjacentes são consecutivos e todo par de vértices consecutivos são adjacentes. Denotamos o *comprimento* de um caminho e de um ciclo, respectivamente, por $e(P)$ e $e(C)$. Seja m esse comprimento, chamamos P_m e C_m de *m-caminho* e *m-ciclo*, respectivamente, chamados de *par* ou *ímpar* de acordo com a paridade de m. Quando um k-ciclo com $k \geq 4$ em um grafo G possui uma aresta em G ligando um par de vértices não consecutivos no ciclo, chamamos essa aresta de *corda*. Definimos como *grafo cordal* aquele em que todo k-ciclo com $k \geq 4$ possui uma corda.

3. Geração de Grafos

3.1. Produto Cartesiano

Definimos a operação de *produto cartesiano* de dois grafos por $G \times H$, um grafo com vértices $V = V(G) \times V(H)$, conforme o produto cartesiano de conjuntos, e arestas $E = \{(u_1, v_1)(u_2, v_2) \mid$ tal que $u_1u_2 \in E(G)$ e $v_1 = v_2$, ou $v_1v_2 \in E(H)$ e $u_1 = u_2\}$. Especificamos o produto cartesiano de dois caminhos $P_n \times P_m$ como uma *$(n \times m)$-grade*.

Calcularemos $P = G \times H$ da seguinte forma: para cada aresta $\{u, v\}$ em $E(G)$, geramos $|V(H)|$ arestas $\{(u,m),(v,m)\}$, $\forall m \in V(H)$, e vice-versa para arestas em $E(H)$. Para montar uma matriz de adjacência optamos por ordenar $V(P) = V(G) \times V(H)$ com maior ordem aos vértices de $V(G)$. Por exemplo, $\langle v_1, v_2\rangle \times \langle u_1, u_2\rangle = \langle (v_1, u_1),$ $(v_1, u_2), (v_2, u_1), (v_2, u_2)\rangle$. A complexidade do algoritmo se dá pela quantidade de vértices e arestas criados para P, ou seja: $\Theta(|E(G)| * |V(H)| + |V(G)| * |E(H)| + |V(G)| * |V(H)|)$. A implementação do produto cartesiano pode ser encontrada em [da Silva 2022].

3.2. Grafos Cordais

Utilizaremos a técnica descrita em [Tarjan and Yannakakis 1984] para gerar grafos cordais a partir de qualquer grafo através da inclusão de arestas. Definimos uma *ordenação total de vértices* do grafo G como $\alpha : V \leftrightarrow \{1, 2, \ldots, n\}$, que atribui, para cada $v \in V$, um número $\alpha(v)$ diferente. Assim, denotaremos por $v_i <_\alpha v_j$ a ordem parcial $\alpha(v_i) < \alpha(v_j)$. Podemos gerar então o *preenchimento produzido pela ordenação* α, $F(\alpha) = \{\{v_i, v_j\} \mid \{v_i, v_j\} \notin E$ e há um caminho de v_i a v_j contendo apenas v_i, v_j e vértices de ordenação inferior a v_i e $v_j\}$. Quando $F(\alpha) = \emptyset$, dizemos ser um *preenchimento zero* e que α é uma *ordenação de preenchimento zero*. O *grafo de eliminação* de G com ordenação α, que possui as arestas de G e seu preenchimento $F(\alpha)$, é denotado por $G(\alpha) = (V, E \cup F(\alpha))$. Representamos com $f(v)$ o *seguidor* de v, que é o seu vizinho com a menor ordenação que respeita $v <_\alpha f(v)$. Recursivamente, usamos $f^i(v)$ para $i \geq 0$, sendo $f^0(v) = v$ e $f^{i+1}(v) = f(f^i(v))$. Dessas definições, seguem os seguintes resultados:

1. $\{v_i, v_j\} \in E \cup F(\alpha) \iff \{v_i, v_j\} \in E$ ou $\exists u$ tal que $\{u, v_i\}, \{u, v_j\} \in E \cup F(\alpha)$, $u <_\alpha v_i$ e $u <_\alpha v_j$.

2. α é uma ordenação de preenchimento zero $\iff$ para todo par de arestas distintas $\{u, v_i\}, \{u, v_j\} \in E$ com $u <_\alpha v_i$ e $u <_\alpha v_j$ então $\{v_i, v_j\} \in E$.

3. α é sempre ordenação de preenchimento zero para seu grafo de eliminação $G(\alpha)$.

4. G é cordal $\iff$ G possui uma ordenação de preenchimento zero.

5. Se α respeita P, então é uma ordenação de preenchimento zero.
   (P) $u <_\alpha v_i <_\alpha v_j, \{u, v_j\} \in E, \{v_i, v_j\} \notin E \implies \exists x, v_i <_\alpha x, \{v_i, x\} \in E, \{u, x\} \notin E$.

6. $\{v, w\} \in E \cup F(\alpha), v <_\alpha w \implies \exists i \geq 1, f^i(v) = w$.

7. $\{v, w\} \in E \cup F(\alpha), v <_\alpha w \iff \exists x, \{x, w\} \in E$ e $f^i(x) = v$, para algum $i \geq 0$.

Logo, ao utilizar α que respeite P, se $F(\alpha) = \emptyset$ então G é cordal. Caso contrário, construímos $G(\alpha)$, que é cordal. Utilizaremos a *busca de máxima cardinalidade*, que respeita P [Tarjan and Yannakakis 1984], para numerar vértices de n a 1 escolhendo sempre o com mais vizinhos numerados. O Algoritmo 1 implementa a busca utilizando $conj(i) = \{v \in V \mid v \text{ tem } i \text{ vizinhos numerados}\}$ para selecionar vértices, e $tam(j)$ para representar a quantidade de vizinhos numerados de v_j, sendo $tam(j) = -1$ para um v_j já numerado. Sempre que numeramos um v_i, para todos seus vizinhos v_j com $tam(j) > -1$, incrementamos $tam(j)$ em 1, retiramos v_j de $conj(k)$ e incluímos $conj(k+1)$. Utilizamos *maior* para salvar o maior i com $conj(i) \neq \emptyset$ e $\alpha^{-1}(k)$ para representar v com numeração k. Como cada vértice e cada aresta é conferida apenas uma vez, a complexidade do Algoritmo 1 é $O(n+m)$.

O Algoritmo 2 gera $G(\alpha)$ processando cada vértice w na ordem crescente de α. Utilizamos $indice(x)$ para armazenar a maior ordem dentre x e seus vizinhos já processados e $f(x)$ armazenar seguidores $f^i(x)$, $i \geq 0$. Seja $ordem = \alpha(w)$, ao processar w, inicializamos $indice(w) \leftarrow ordem$ e $f(w) \leftarrow w$, pois ainda não passamos por um vértice de maior ordem. Então, para todo vizinho v de w tal que $v <_\alpha w$, utilizamos o resultado 7: para todo $f^i(v)$, $i \geq 0$, tal que $indice(f^i(v)) < ordem$, inserimos $\{w, f^i(v)\}$ em $E \cup F(\alpha)$ e atualizamos $indice(f^i(v)) \leftarrow ordem$, pois w é o vizinho de maior ordem. Por fim, caso o último seguidor pelo qual passamos tenha a si mesmo como seguidor, atribuímos w como seu seguidor, pois após conectarmos w este se torna o vizinho de maior ordem. Como passamos por cada vértice e aresta apenas uma vez, a complexidade do Algoritmo 2 é $O(n+m')$, com $m' = |E \cup F(\alpha)|$.

Com os Algoritmos 1 e 2, implementamos as funções de checar se um grafo é cordal, retornando falso se $F(\alpha) \neq \emptyset$, de tornar cordal um grafo qualquer, inserindo $F(\alpha)$ em E, e de gerar um grafo cordal a partir de um k-ciclo, que recebe apenas o número de vértices $k \geq 0$. Todas as funções executam ambos os algoritmos em sequência, portanto possuindo complexidade $O(n+m')$.

Algoritmo 1: BUSCAMAXCARD

Entrada: G
Saída: α e α^{-1}

para i *de* 0 *a* $n-1$ **faça**
  $conj(i) \leftarrow \emptyset$;
para $v \in V(G)$ **faça**
  $tam(v) \leftarrow 0$;
  adiciona v **ao** $conj(0)$;
$numero \leftarrow n$;
$maior \leftarrow 0$;
enquanto $numero \geq 1$ **faça**
  enquanto $maior \geq 0$ *e* $conj = \emptyset$ **faça**
    $maior \leftarrow maior - 1$;
  $v \leftarrow$ **remove algum do** $conj(maior)$;
  $\alpha(v) \leftarrow numero$;
  $\alpha^{-1}(numero) \leftarrow v$;
  $tam(v) \leftarrow -1$;
  para $w \in N(v)$ *tal que* $tam(w) \geq 0$ **faça**
    remove w **do** $conj(tam(w))$;
    $tam(w) \leftarrow tam(w) + 1$;
    adiciona w **ao** $conj(tam(w)$;
  $numero \leftarrow numero - 1$;
  $maior \leftarrow maior + 1$;
retorna α *e* α^{-1};

Algoritmo 2: PREENCHIMENTO

Entrada: E, α, α^{-1}
Saída: $E \cup F(\alpha)$

para *ordem de* 1 *a* n **faça**
  $w \leftarrow \alpha^{-1}(ordem)$;
  $f(w) \leftarrow w$;
  $indice(w) \leftarrow ordem$;
  para $v \in N(w)$ *em* $E \cup F(\alpha)$ *com* $\alpha(v) < ordem$ **faça**
    $x \leftarrow v$;
    enquanto $indice(x) < ordem$ **faça**
      /* Se entra no ciclo, G não é cordal */
      $F(\alpha) \leftarrow F(\alpha) \cup \{x, w\}$;
      $indice(x) \leftarrow ordem$;
      $x \leftarrow f(x)$;
    se $f(x) = x$ **então** $f(x) \leftarrow w$;
retorna $E \cup F(\alpha)$;

4. Problemas em Grafos

4.1. Clique Máxima e Conjunto Independente Máximo

O conjunto $C \subseteq V$ é uma *clique* de G se, e somente se, para cada $u, v \in C, \exists\{u, v\} \in E$, enquanto $S \subseteq V$ é um *conjunto independente* de G se, e somente se, para cada $u, v \in S, \nexists\{u, v\} \in E$. Definimos os problemas da *clique máxima* e do *conjunto independente máximo* como a busca pelos C e S com mais vértices, respectivamente. A cardinalidade dos C e S máximos em um grafo G é denotada por $\omega(G)$ e $\alpha(G)$, respectivamente. Encontrar estas cardinalidades é tão difícil quanto encontrar os conjuntos em si [Bondy and Murty 2008]. Caso haja múltiplos C ou S com $\omega(G)$ e $\alpha(G)$, respectivamente, qualquer um é uma solução válida. Como $\exists\{u, v\} \in G \iff \nexists\{u, v\} \in \bar{G}$, logo, um conjunto $X \subseteq V$ é um conjunto independente de $G \iff X$ é uma clique de $\bar{G}$ e $\omega(G) = \alpha(\bar{G})$. Portanto, como os problemas são redutíveis entre si e o *3-SAT* reduz à clique máxima, ambos são *NP-Difíceis* [Bondy and Murty 2008]. Optamos por utilizar uma variação dos algoritmos de [Robson 1986] para solucionar o conjunto independente máximo e aplicá-la em $\bar{G}$ resolvendo também a clique máxima, já que podemos gerar $\bar{G}$ em tempo $O(n^2)$.

Denotamos por $N[v] = \{v\} \cup N(v)$ a *vizinhança fechada* de v; $N^2(v)$ o conjunto da união $\bigcup_{w \in N(v)} N(w)$, mas que exclui v; $N(w)$, $\forall w \in N(v)$, excluindo v; e $G \setminus u$ o subgrafo induzido de G após remover o vértice u. Seja S um conjunto independente máximo de G e $v \in V$ um vértice qualquer: ou $v \in S$, logo $N(v) \cap S = \emptyset$ ou $v \notin S$, portanto seus vizinhos podem estar em S. Ademais, qualquer $X \subseteq S$ também é um conjunto independente. Isso nos leva à recursão presente na Equação 1.

$$\alpha(G) = \max\{\alpha(G \setminus v),\ 1 + \alpha(G \setminus N[v])\}. \tag{1}$$

Perceba que $S \cap N[v] \neq \emptyset$, $\forall v \in V$, pois se $S \cap N(v) = \emptyset$, então $v \in S$, enquanto, se $v \notin S$, ao menos um vértice de $N(v)$ está em S. Além disso, se apenas um vizinho $u \in N(v)$ está em S, o tamanho do conjunto não se altera, portanto damos preferência a incluir o próprio v. Com isso, podemos restringir mais o caso em que $v \notin S$ e reescrever $\alpha(G \setminus v)$ com a função $\alpha^2(G \setminus v, N(v))$, que retorna o $\alpha(G \setminus v)$ que possui ao menos dois vizinhos de v, como na Equação 2.

$$\alpha(G) = \max\{\alpha^2(G \setminus v,\ N(v)),\ 1 + \alpha(G \setminus N[v])\}. \tag{2}$$

Então escolhemos a Equação 1 quando o grau for alto, pois $\alpha(G \setminus N[v])$ será suficientemente mais fácil de resolver, enquanto optamos pelo cálculo com a Equação 2 para graus baixos, pois a análise dos vizinhos acelera a solução de $\alpha(G \setminus v)$. Utilizamos [Robson 1986] para decidir $d(v) \geq 8$ como um grau alto e $d(v) < 8$ como um grau baixo.

Dizemos que v *domina* u se $N[v] \subset N[u]$. Incluir um vértice dominado em S impede que os vizinhos do dominador e outros vértices estejam em S, portanto $\alpha(G \setminus u) \geq \alpha(G \setminus v)$. Escolhendo incluir sempre o vértice dominante, temos que $\alpha(G) = \alpha(G \setminus u)$. Estendemos a definição para conjuntos independentes $A \subseteq V$ e $B \subseteq V$: A domina B quando $|A| \geq |B|$ e $\bigcup_{a \in A} N[a] \subset \bigcup_{b \in B} N[b]$, sendo a melhor opção para construir o conjunto independente máximo.

Nos referenciaremos aos algoritmos para encontrar o conjunto independente máximo por *cim*. Assumiremos que G está acessível globalmente e passaremos um conjunto independente $X \subseteq V$ como entrada para ser expandido. O Algoritmo 3 retorna $cim(X)$. Para $|X| \leq 2$ é trivial. Para X desconexo, apenas retornamos a união dos *cim* de suas componentes conexas. Para $|X| > 2$, escolhemos v com $d(v)$ minimal e um $u \in N(v)$ com $d(u)$ maximal. Portanto, $d(v) \leq d(u)$. Separamos os casos $d(v) = 1$, 2 e 3.

Algoritmo 3: $cim(X)$

Entrada: X
Saída: Um conjunto independente máximo de X em G

se $|X| = 0$ **então**
  retorna $\emptyset$;
se *NãoConexoEmX(G,X)* **então** `// Testa se X é não-conexo em G`
  $C \leftarrow$ MenorConexo(G,X) `// Retorna menor componente conexo de X`;
  retorna $cim(X \setminus C) \cup cim(C)$;
se $|X| \leq 2$ **então**
  Escolhe $v \in X$;
  retorna $\{v\}$;
Escolhe $v \in X$ com $d(v)$ minimal;
Escolhe $u \in N(v)$ com $d(u)$ maximal;
se $d(v) = 1$ **então retorna** $\{v\} \cup cim(X \setminus N[v])$;
se $d(v) = 2$ **então**
  Escolhe $u' \in N(v) \setminus \{u\}$;
  se $\{u, u'\} \in E(G)$ **então retorna** $\{v\} \cup cim(X \setminus N[v])$;
  retorna *MaiorConj* $(\{u, u'\} \cup cim(X \setminus \{N[u] \cup N[u']\}), \{v\} \cup cim^2(X \setminus N[v], N^2(v)))$;
se $d(v) = 3$ **então retorna** *MaiorConj* $(cim^2(X \setminus \{v\}, N(v)), \{v\} \cup cim(X \setminus N[v]))$;
se v *domina* u **então retorna** $cim(X \setminus \{u\})$;
retorna *MaiorConj* $(cim(X \setminus \{u\}), \{u\} \cup cim(X \setminus N[u]))$;

Somente para $d(v) = 2$ se faz necessária uma explicação mais clara: denotamos o outro vizinho de v por u'. Caso $\{u,u'\} \in E$, então vuu' é um $3-ciclo$ em que v domina ambos os vértices, portanto retornamos $\{v\} \cup cim(X \setminus N[v])$. Caso $\{u,u'\} \notin E$, escolhemos a melhor dentre as duas opções: retornar $\{u,u'\} \cup cim(X \setminus \{N[u] \cup N[u']\})$, ou $\{v\} \cup cim^2(X \setminus N[v], N^2(v))$. O Algoritmo 4 implementa o cim^2, que retorna o $cim(X)$ com ao menos dois vértices de $N^2(v)$, necessários para ser uma melhor opção que a anterior. Para $d(v) > 3$, caso v domine u, retornamos $\{v\} \cup cim(X \setminus \{u\})$, caso não, como não há forma de u dominar v, analisamos as opções dentre $\{u\} \cup cim(X \setminus N[u])$ e $cim(X \setminus \{u\})$ e retornamos o maior conjunto.

Algoritmo 4: $cim^2(X,S)$

Entrada: $X, S = \{s_1, s_2, \ldots\}$ com $d(s_i) \leq d(s_{i+1})$
Saída: Conjunto Independente Máximo de X em G com ao menos dois elementos de S

se $|S| \leq 1$ **então retorna** $\emptyset$;
se $|S| = 2$ **então**
  se $\{s_1, s_2\} \in E(G)$ **então retorna** $\emptyset$;
  retorna $cim(X \setminus \{N[s_1] \cup N[s_2]\}) \cup \{s_1, s_2\}$;
se $|S| = 3$ **então**
  se $d(s_1) = 0$ **então retorna** $cim^1(X \setminus \{s_1\},\ S \setminus \{s_1\}) \cup \{s_1\}$;
  se $\{s_1, s_2\}, \{s_2, s_3\}, \{s_3, s_1\} \in E(G)$ **então retorna** $\emptyset$;
  se $\{s_i, s_j\}, \{s_i, s_k\} (j \neq k) \in E(G)$ **então retorna** $cim(X \setminus \{N[s_j] \cup N[s_k]\}) \cup \{s_j, s_k\}$;
  se $\{s_i, s_j\} \in E(G)$ **então retorna** $cim^1(X \setminus N[s_k],\ [s_i, s_j]) \cup \{s_k\}$ $(i \neq k \neq j)$;
  se $\exists v \in N(s_i) \cap N(s_j) (i \neq j)$ **então retorna** $cim^2(X \setminus \{v\},\ S)$;
  se $d(s_1) = 1$ **então retorna** $cim^1(X \setminus N[s_1],\ S \setminus \{s_1\}) \cup \{s_1\}$;
  retorna *MaiorConj(*$cim^1(X \setminus N[s_1],\ S \setminus \{s_1\}) \cup \{s_1\}$,
    $cim^2(X \setminus \{N[s_2] \cup N[s_3] \cup \{s_1\}\},\ N(s_1))$*)*;
se $|S| = 4$ **então**
  se $\exists v \in X$ *com* $d(v) \leq 3$ **então retorna** $cim(X)$;
  retorna *MaiorConj(*$cim(X \setminus N[s_1]) \cup \{s_1\}$, $cim^2(X \setminus \{s_1\},\ S \setminus \{s_1\})$*)*;
retorna $cim(X)$;

Algoritmo 5: $cim^1(X,S)$

Entrada: $X, S = \{s_1, s_2\}$ com $d(s_1) \leq d(s_2)$
Saída: Conjunto Independente Máximo de X em G que inclui apenas um elemento de S

se $d(s_1) \leq 1$ **então retorna** $cim(X)$;
se $\{s_1, s_2\} \in E(G)$ **então**
  se $d(s_1) \leq 3$ **então retorna** $cim(X)$;
  retorna *MaiorConj(*$cim(X \setminus N[s_1]) \cup \{s_1\}, cim(X \setminus N[s_2]) \cup \{s_2\}$*)*;
se $N(s_1) \cap N(s_2) \neq \emptyset$ **então retorna** $cim^1(X \setminus \{N(s_1) \cap N(s_2)\}, S)$;
se $d(s_2) = 2$ **então**
  $e, f \leftarrow$ os elementos de $N(s_1)$;
  se $\{e, f\} \in E(G)$ **então retorna** $cim(X \setminus N[s_1]) \cup \{s_1\}$;
  se $\{N(e) \cup N(f)\} \setminus \{s_1\} \subset N(s_2)$ **então retorna** $cim(X \setminus \{N[s_1] \cup N[s_2]\}) \cup \{e, f, s_2\}$;
  retorna *MaiorConj(*$cim(X \setminus N[s_1]) \cup \{s_1\}$, $cim(X \setminus \{N[e] \cup N[f] \cup N[s_2]\}) \cup \{e, f, s_2\}$*)*;
retorna *MaiorConj(*$cim(X \setminus N[s_2]) \cup \{s_2\}$, $cim^2(X \setminus (N[s_1] \cup \{s_2\}),\ N(s_2)) \cup \{s_1\}$*)*;

O Algoritmo 4 leva em consideração que cim^2 só é chamado para X conexo, com $|X| \geq 3$ e $d(v) = 3$ ou 2, não podendo formar um $3-ciclo$ neste último caso. Para três casos em que $|X| = 3$, utilizamos a função auxiliar cim^1, implementada no Algoritmo 5,

com a seguinte lógica: para um vértice $v \in X$, podemos garantir que ou v e mais um vértice de X estão no conjunto máximo independente S, ou $v \notin S$, mas há dois outros vértices de X em S. A função cim^1 realiza essa escolha recebendo um conjunto X' e dois vértices s_1, s_2 para retornar o maior conjunto independente de X que contém apenas s_1 ou s_2.

A geração do grafo complementar em tempo polinomial não afeta a complexidade exponencial dos Algoritmos 3, 4 e 5. Portanto, tanto para a resolução do conjunto independente máximo quanto da clique máxima, o tempo dos algoritmos anteriores em conjunto é $O(2^{0.276n})$, conforme [Robson 1986].

4.2. Emparelhamento Máximo

Dizemos que $M \subseteq E(G)$ é um *emparelhamento* de G se, e somente se, qualquer par de arestas em M não incide sobre um mesmo vértice. Encontrar o M de maior cardinalidade é o problema do *emparelhamento máximo*.

Em [Hopcroft and Karp 1973] é apresentado um algoritmo polinomial que utiliza uma estratégia gulosa para construir um emparelhamento máximo em tempo $O(n^{\frac{5}{2}})$ baseando-se nos seguintes conceitos: seja M um emparelhamento sobre G, dizemos que $v \in V(G)$ é um *vértice livre* se, e somente se, nenhuma aresta de M incide sobre ele. Um *caminho alternante* em $A \subseteq E$ é um caminho cuja sequência de arestas $\langle e_1, \ldots, e_m \rangle$ alterna entre $e_i \in A$ e $e_{i+1} \notin A$. Dizemos que P é um *caminho aumentativo* de M se, e somente se, os vértices em suas extremidades são livres e ele é alternante em M. Note que, trocando as arestas de P que não estão em M com as que estão, obtemos um novo emparelhamento $M \oplus P$, $\oplus$ a diferença simétrica, em que $|M \oplus P| = |M| + 1$, não importando o comprimento de P. O algoritmo em [Hopcroft and Karp 1973], ilustrado na Figura 1, itera realizando essa ação de *aumentação* sucessivamente, utilizando o teorema de que M é um emparelhamento máximo se, e somente se, não possui caminho aumentativo.

Figura 1. Arestas em vermelho pertencem ao emparelhamento. Vértices verdes são livres. Construímos o caminho aumentativo $P = (4,2),(2,7),(7,6)$ no grafo (a) e realizamos a aumentação $M \oplus P = \{(1,3),(2,4),(7,6)\}$, obtendo o emparelhamento no grafo (b), que é máximo, pois não há como construir um caminho aumentativo nele.

Seja $\{M_i\}$ uma sequência de emparelhamentos em que $M_{i+1} = M_i \oplus P_i$ para P_i o caminho aumentativo mais curto em M_i, [Hopcroft and Karp 1973] demonstra que:

1. $|P_i| \leq |P_{i+1}|$.
2. $|P_i| = |P_j| \implies P_i$ e P_j não possuem vértices em comum.

3. Seja s a cardinalidade do emparelhamento máximo sobre G, a sequência de cardinalidades $|P_1|, |P_2|, \ldots, |P_i|, \ldots$ possui, no máximo, $2\lfloor\sqrt{s}\rfloor + 2$ números distintos.

Por 1 e 2, computamos todos os caminhos aumentativos minimais $P_1, P_2, \ldots, P_k$, sem vértices em comum, referentes a um mesmo M e atribuímos $M \leftarrow M \oplus P_1 \oplus P_2 \oplus \ldots \oplus P_k$ para gerar um emparelhamento de cardinalidade $|M| + k$. O limite de vezes que realizaremos essas atribuições está definido em 3. Encontramos o emparelhamento máximo pelo Algoritmo 6.

Algoritmo 6: EMPARELHAMENTOMAX

Entrada: G
Saída: Emparelhamento Máximo de G

```
1 M ← ∅;
2 enquanto PossuiCaminhoAumentantivo(M) faça
3     P1, P2, ..., Pk ← CaminhosAumentantivosMinimais(M);
4     M ← M ⊕ P1 ⊕ P2 ⊕ ... ⊕ Pk;
5 retorna M;
```

Encontrar o conjunto maximal de caminhos aumentativos minimais com relação a M é um passo custoso com várias propostas de solução na literatura. Optamos pela abordagem da *contração de blossom* [Shoemaker and Vare 2016].

Construímos uma *floresta* F, em que cada *árvore* tem um vértice livre como raiz. Seja v um vértice livre, para toda aresta $\{v,w\} \in E$, w é filho de v em F. Se w for livre, encontramos um caminho aumentativo. Senão existe $\{w,x\} \in M$, logo adicionamos x como filho de w. Repetimos esse processo para todo vértice de F, analisando apenas arestas fora de M. Dessa forma, o caminho mais curto de qualquer vértice em uma árvore de F até sua raiz será um caminho alternante. Dividimos a análise de um vértice $v \in F$ e uma nova aresta $\{v,w\} \notin M$ nos quatro casos abaixo. A Figura 2 ilustra os três principais.

1. w não está na floresta;

2. w está na floresta, mas sua distância até sua raiz é ímpar;

3. w está na floresta, sua distância até sua raiz é par e sua raiz não é a raiz de v;

4. w está na floresta, sua distância até sua raiz é par e sua raiz é a mesma de v.

Para o caso 1, apenas adicionamos w como filho de v e x como filho de w, para $\{w,x\} \in M$. No caso 2, seja p o pai de w, $\{p,w\} \notin M$, implicando no caminho $\langle v,w,p \rangle$ não ser alternante, portanto a aresta é ignorada. Para o caso 3, a distância par de w à sua raiz nos permite construir um caminho aumentativo entre as raízes de v e w passando por $\{v,w\}$. No caso 4, incluir $\{v,w\}$ não forma um caminho aumentativo, mas um ciclo alternante de comprimento ímpar passando por v, w e a raiz de ambos, chamado *blossom*. A solução em [Shoemaker and Vare 2016] é de "contrair"o *blossom* conforme a Figura 3, tornando-o um único vértice para manter a estrutura da árvore de caminhos alternantes, mas sem ciclos.

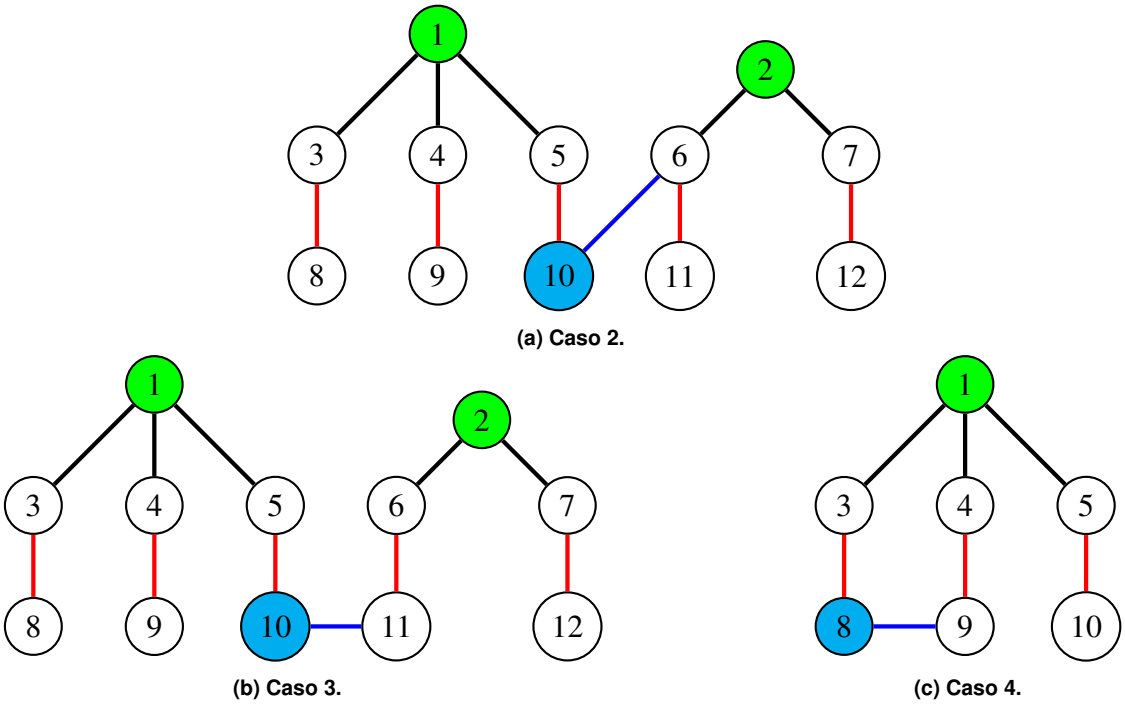


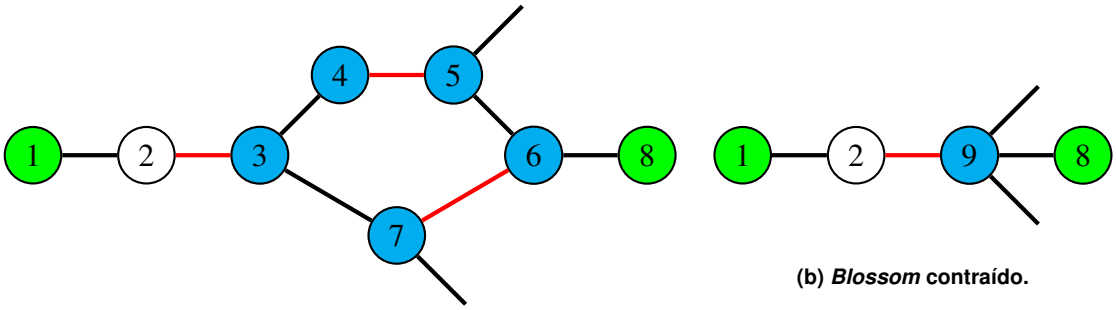
Figura 2. Arestas em vermelho pertencem ao emparelhamento. Vértices verdes são livres. A aresta e o vértice em análise na iteração estão em azul. Em (b) encontramos um caminho aumentativo entre 1 e 2. Isso não ocorre em (a), pois a distância de 6 à sua raiz é impar, logo o caminho não alterna. Em (c) a inclusão de $\{8,9\}$ forma um ciclo alternante: $\langle 1,3,8,9,4 \rangle$.

Algoritmo 7: CAMINHOAUMENTATIVO

Entrada: G, M
Saída: Caminho aumentativo em G com relação a M

```
F ← floresta vazia ;
Fila ← vértices livres de G em M ;
para v ∈ Fila faça        // Cada vértice livre é uma raiz de árvore na floresta
    InsereRaiz(F, v);
    Raiz(v) ← v;
ArestasNovas ← E \ M;
para v ∈ Fila faça
    para {v,w} ∈ ArestasNovas faça
        se w ∉ F então                // Caso 1:  vértice fora da floresta
            AdicionaNaFloresta(M,F,v,w) ;
            Fila ← Fila ∪ {w};
        senão se Distância(w, raiz(w)) % 2 == 0 então
            se raiz(v) ≠ raiz(w) então        // Caso 3:  caminho entre duas raízes
                retorna RetCaminhoAumentativo(F,v,w);
            senão                         // Caso 4:  ciclo na mesma árvore
                retorna CaminhoBlossom(G,M,F,v,w);
        ArestasNovas ← ArestasNovas \ {{v,w}};
        Fila ← Fila \ {v};
retorna CaminhoVazio();
```

Obtido o caminho aumentativo P, verificamos se há blossons contraídos nele. Caso não, P é a solução. Caso sim, analisamos o *blossom* B como o ciclo $\langle r,\ldots,v,w,\ldots\rangle$, em que r é a raiz da sub-árvore com os ramos que levam a v e w, e sua posição em P, para *expandir* B e obter o caminho P_B sem o *blossom*. Chamamos de *haste* a aresta $\{x,B\}$ em P. Toda haste foi contraída de uma *aresta original* $\{x,y\}$, tal que $y \in B$. Se $P = \langle\ldots,a,B,c,\ldots\rangle$, então $\{a,B\} \in M$ e $\{B,c\} \notin M$, ou o contrário. Nesse caso, geramos P_B substituindo B pelo caminho de comprimento par $\langle x,\ldots,y\rangle \in B$, tal que $\{a,x\}$ e $\{y,c\}$ são arestas originais. Se $P = \langle B,a,\ldots\rangle$, por ser um caminho aumentativo, $\{B,a\} \notin M$. Então substituímos B por $\langle r,\ldots,y,x,a\rangle$, sendo $\{x,a\}$ aresta original e $\{y,x\} \in M$. O mesmo vale para $P = \langle\ldots,a,B\rangle$, que substituímos por $\langle a,x,y,\ldots,r\rangle$. Caso P_B ainda possua um *blossom*, repetimos o processo.



(a) *Blossom* expandido.

(b) *Blossom* contraído.

Figura 3. Arestas em vermelho se referem ao emparelhamento. Vértices verdes estão livres e azuis pertencem ao *blossom*. Contraímos o *blossom* em (a) gerando o grafo (b). Perceba que as arestas de vértices do *blossom* são mantidas como arestas de 9, o *blossom* contraído.

Algoritmo 8: ADICIONANAFLORESTA

Entrada: M,F,v,w
Saída: Altera a floresta F

InsereAresta($\{v,w\}$);
Raiz(w) $\leftarrow$ Raiz(v);
`/* Inserimos o emparelhado a w */`
$x \leftarrow$ vértice emparelhado com w em M;
InsereAresta($\{w,x\}$);
Raiz(x) $\leftarrow$ Raiz(v);

Algoritmo 9: RETCAMINHOAUMENTATIVO

Entrada: F,v,w
Saída: Caminho aumentativo entre as raízes de v e w

`/* CamCurto(F,x,y) é o caminho mais curto de x a y em F */`
CamV $\leftarrow$ CamCurto($F, Raiz(v), v$);
CamW $\leftarrow$ CamCurto($F, w, Raiz(w)$);
retorna *Concatena(CamV, CamW)*;

Algoritmo 10: CAMINHOBLOSSOM

Entrada: G,M,F,v,w
Saída: Caminho aumentativo em G com relação a M

$B \leftarrow Concatena(CamCurto(F,v,w),[v])$; `// Blossom B`
$G' \leftarrow G$ com B contraído;
$M' \leftarrow M$ com B contraído;
$Caminho' \leftarrow CaminhoAumentativo(G',M')$;
se $w \in P'$ **então retorna** P' *com B expandido* ;
senão retorna P' ;

Portanto, utilizando o Algoritmo 7 e suas sub-rotinas, os Algoritmos 8, 9 e 10, para encontrar caminhos aumentativos e realizando a aumentação do Algoritmo 6, conseguimos resolver o problema do emparelhamento máximo em tempo $O(n^2m)$, como prova [Shoemaker and Vare 2016].

5. Considerações Finais

Os códigos dos algoritmos para produto cartesiano, complemento não foram apresentados neste texto, porém estão implementados em [da Silva 2022], assim como todos os algoritmos presentes no artigo. Ao final, implementamos as funções de checar cordabilidade, encontrar clique, conjunto independente e emparelhamento máximos, gerar ciclo cordal e grade, realizar o produto cartesiano de dois grafos e tornar qualquer grafo cordal. O trabalho foi desenvolvido em cooperação com o doutorando Braully Rocha da Silva, expandindo a plataforma *web* de problemas em grafos descrita em [da Silva 2018] para agregar as funções citadas acima. Todos os códigos foram escritos em *C* ou *C++*, com alguns também em *Java* para integrar ao sistema, cujos códigos e servidor de aplicação estão disponíveis em [git 2022]. Como sugestão para trabalhos futuros, propomos a adição do reconhecimento da família de *grafos fortemente regulares* e a solução de problemas geralmente NP-Difíceis, como a clique máxima, por algoritmos polinomiais através da restrição a determinadas famílias de grafo.

Referências

(2022). Graph problems tool. `https://github.com/braully/graph-problems-tool`. Acessado em 15 de Agosto de 2022.

Bondy, J. and Murty, U. (2008). Graph theory, 6 springer. *Grad. Texts in Math*, 244.

da Silva, B. R. (2018). Algoritmos e limites para os números envoltório e de carathéodory na convexidade p_3.

da Silva, D. C. (2022). Algoritmos e Implementações em Grafos. `https://github.com/DanteCampos/IC_2021-2022_Graphs/`. Acessado em 15 de Agosto de 2022.

Halim, S. (2015). Visualgo–visualising data structures and algorithms through animation. *Olympiads in informatics*, 9:243–245.

Hopcroft, J. E. and Karp, R. M. (1973). An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231.

Robson, J. M. (1986). Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425–440.

Shoemaker, A. and Vare, S. (2016). Edmonds' blossom algorithm. *CME*.

Tarjan, R. E. and Yannakakis, M. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on computing*, 13(3):566–579.

UnickSoft (2022). Graph online. `https://graphonline.ru/en/`. Acessado em 15 de Agosto de 2022.