Aluno: Leonardo Bertucci dos Santos
Orientador: Cristiane Maria Sato

1. Grafos

1.1. Grafos

Um **grafo** X consiste de um conjunto de **vértices** $V(X)$ e um conjunto de **arestas** $E(X)$ tal que uma aresta é um par não ordenado de elementos de $V(X)$. Denotamos uma aresta (u,v) simplesmente por uv. Se $e = uv$ é uma aresta de X dizemos que u e v são **adjacentes** ou **vizinhos**, e escrevemos $u \sim v$; dizemos também que e **liga** os vértices u e v, e que **incide** sobre cada um deles. Um grafo é dito **completo** se todos os seus vértices são adjacentes, sendo denotado K_n o grafo completo com n vértices, e **vazio** se não possui nenhuma aresta (mas pelo menos um vértice). O grafo sem vértices nem arestas é chamado **grafo nulo**.

Grafos da forma que definimos aqui são chamados as vezes de **grafos simples**, pois existem algumas definições mais gerais que permitem por exemplo arestas paralelas ou loops (aresta de um vértice a si mesmo). Uma generalização importante ocorre se considerarmos pares ordenados de $V(X)$, denominados **arcos** ou **arestas dirigidas**, no lugar das arestas. Definimos desta forma um **grafo dirigido**, ou **digrafo**, como $V(X)$ junto com um conjunto de arcos $A(X)$. Podemos ver nesse contexto um grafo simples como um grafo dirigido onde (v,u) é um arco sempre que (u,v) for um arco. Mencionaremos neste texto sempre que formos trabalhar com digrafos ou qualquer outra variação de grafo, e caso contrário usaremos a palavra "grafo" sempre para identificar um grafo simples com conjunto de vértices finito.

1.2. Subgrafos

Um **subgrafo** de um grafo X é um grafo Y tal que

$$V(Y) \subseteq V(X), \quad E(Y) \subseteq E(X).$$

Se $V(Y) = V(X)$, dizemos que Y é **subgrafo gerador** de X; se $V(Y) \neq V(X)$ então Y é um **subgrafo próprio**. Y é um **subgrafo induzido** se dois vértices em $V(Y)$ são adjacentes se e somente se eles são adjacentes em $V(X)$. Um subgrafo gerador pode ser obtido deletando-se algumas arestas de X, enquanto um subgrafo induzido pode ser obtido ao se deletar alguns vértices de X (junto com as arestas que se ligavam a eles).

1.3. Homomorfismos

Definição 1. Sejam X, Y grafos. Uma função $f : V(X) \to V(Y)$ é um **homomorfismo** se $f(u)$ e $f(v)$ são adjacentes sempre que u é adjacente a v.

Um homomorfismo de X em si mesmo é um **endomorfismo**. O conjunto de todos os endomorfismos em um grafo X forma um monoide (estrutura algébrica com operação binária associativa e que possui elemento neutro). Uma propriedade interessante que pode ser caracterizada por meio de homomorfismos é o número cromático de um grafo: uma k-**coloração própria** de uma grafo X é uma função de $V(X)$ em um conjunto de k cores tal que vértices adjacentes são levados em cores diferentes. O menor número k para o qual X pode ser própriamente k-colorido é chamado **número cromático** de X, e é denotado por $\chi(X)$. O conjunto de vértices com uma determinada cor forma um conjunto independente.

Lema 1. *O número cromático de um grafo X, $\chi(X)$, é igual ao menor inteiro r tal que existe um homomorfismo de X para K_r.*

Definição 2. Uma **retração** é um homomorfismo de um grafo X em um subgrafo Y de si mesmo tal que a restrição $f \upharpoonright_Y$ de f para $V(Y)$ é a identidade. Se existe uma retração de X para um subgrafo Y, dizemos também que Y **é uma retração de** X.

De fato, se existe $f : X \to Y$ tal que $f \upharpoonright_Y$ seja uma bijeção, então Y será uma retração de X via a função $g = (f \upharpoonright_Y)^{-1} \circ f$. Logo, uma retração de X é um subgrafo Y para o qual X possui um homomorfismo sobrejetor.

Definição 3. Um **isomorfismo** ϕ de X em Y é um homomorfismo bijetivo cuja inversa é também um homomorfismo. Se X e Y são isomorfos, escrevemos $X \cong Y$.

Em outras palavras, ϕ é um isomorfismo quando $f(u)$ e $f(v)$ são adjacentes se e somente se u e v são adjacentes. Dois grafos isomorfos possuem exatamente a mesma estrutura e em geral podemos tratá-los como se fossem iguais.

Um isomorfismo de X em si mesmo é chamado um **automorfismo**. O conjunto de todos os automorfismos em um grafo X forma um grupo, denominado **grupo de automorfismos** de X e denotado como $Aut(X)$. O grupo de automorfismos de um grafo X é um subgrupo do grupo simétrico $Sym(V(X))$, o grupo de todas as permutações dos vértices de X. Se X possui n vértices, escreveremos $Sym(n)$ ao invés de $Sym(V(X))$. Em particular, $Aut(K_n) \cong Sym(n)$, já que toda permutação de vértices no grafo completo é um automorfismo.

Definimos uma relação $\to$ na classe de todos os grafos por $X \to Y$ se existe homomorfismo de X em Y. Como a composição de homomorfismos é um homomorfismo, $\to$ é transitiva; $\to$ é também reflexiva já que a identidade é um homomorfismo. Não é difícil ver que nossa nova relação não é simétrica nem anti-simétrica, e $\to$ é assim uma pré-ordem na classe de todos os grafos. Chamaremos $\to$ de **pré-ordem de homomorfismos**. Dois grafos que não admitem homomorfismo de um no outro são ditos **incomparáveis**, e caso contrário, são **comparáveis**. Dois grafos X e Y tais que $X \to Y$ e $Y \to X$ são ditos **homomorficamente equivalentes**.

Se f é um homomorfismo de X para Y, as pré imagens $f^{-1}(y)$ de cada vértice $y \in Y$ determinam uma partição π de $V(X)$ chamada **kernel de** f e denotada por $\ker f$. O kernel de f é uma partição em conjuntos independentes. Dado um grafo X e uma partição π de $V(X)$, definimos um grafo X/π tomando as classes de π como vértices e uma aresta entre duas classes se existe uma aresta em X conectando estas classes. Existe um homomorfismo natural de X em X/π com kernel π. Note que X/π será um grafo simples se e somente se π for uma partição de $V(X)$ em conjuntos independentes.

1.4. Cores

Definição 4. Um grafo X é um **core** (ou **núcleo**) se todo homomorfismo de X em si mesmo é uma bijeção. Um subgrafo Y de X é um **core de** X se Y é um core e existe um homomorfismo de X para Y ($X \to Y$). Denotamos o core de X por $X^\bullet$.

1.5. Algoritmos

Nesta seção exibiremos algoritmos que foram desenvolvidos ao longo do trabalho acompanhando o estudo dos temas e poderemos discutir sobre a complexidade assintótica de alguns desses problemas, trazendo resultados que sejam relevantes. Os códigos foram feitos utilizando a linguagem python e utilizamos a representação matricial para representar grafos no computador.

Começamos verificando a existência de um homomorfismo de um grafo X em um grafo Y e depois se são isomorfos:

```
def verify_homo(X, Y):
    for f in gen_func(len(X), len(Y)):
        homo = True
        # vejamos se f é homomorfismo:
        for (i, j) in edges(X):
            if Y[f[i]][f[j]] == 0:
                homo = False
                break
        if homo: return f
    return False
```

```
def verify_iso(X, Y):
    if len(X) != len(Y): return False
    for f in list(itertools.permutations(list(range(len(X))))):
        iso = True
        # vejamos se f é homomorfismo:
        for (i, j) in edges(X):
            if Y[f[i]][f[j]] == 0:
```

```
                iso = False
                break
        if iso:  # vejamos se f^-1 é homomorfismo:
            g = invert(f)
            for (i, j) in edges(Y):
                if X[g[i]][g[j]] == 0:
                    return False
        if iso: return f
    return False
```

Aqui a função *gen_func* gera todas as funções dos vértices de X nos vértices de Y, dando caráter exponencial ao algoritmo *verify_homo*. De fato, o caso geral do problema de se dizer se existe um homomorfismo de um grafo X em um grafo Y é NP-completo (citar fonte?).

Para $f : X \to Y$ ser um isomorfismo, f deve ser uma bijeção. Assim, olhamos agora apenas para as permutações em n elementos para gerar as funções canditatas em *verify_iso*. O problema se mantém com tempo de execução exponencial para o pior caso. Entretanto, ainda não se foi possível mostrar que o problema é NP-completo, sendo um grande canditato a membro da classe de problemas NP que não se encontram em P nem em NP-completo [1]. Uma implementação otimizada para esse problema, assim como para encontrar o grupo de automorfismos de um grafo, é o progrma *nauty*, de Brendan McKay, que consegue resolvê-lo para grafos com grande número de vértices [2].

```
# Listando homomorfismos de X em Y
def list_homo(X, Y):
    lista_homo = []
    for f in gen_func(len(X), len(Y)):
        homo = True
        # vejamos se f é homomorfismo:
        for (i, j) in edges(X):
            if Y[f[i]][f[j]] == 0:
                homo = False
                break
        if homo: lista_homo.append(f)
    return lista_homo
```

```
# Listando todos os automorfismos de X
def list_aut(X):
    lista_auts = []
    for f in list(itertools.permutations(list(range(len(X))))):
        iso = True
        for (i, j) in edges(X):
            if X[f[i]][f[j]] == 0:
                iso = False
```

```
                break
        if iso:
            g = invert(f)
            for (i, j) in edges(X):
                if X[g[i]][g[j]] == 0:
                    iso = False
                    break
        if iso: lista_auts.append(f)
    return lista_auts
```

a

A. Classes de Complexidade

Referências Bibliográficas

[1] Godsil, C. Royle, G. *Algebraic Graph Theory*, Springer, 2001.

[2] McKay, B. D. *nauty User's Guide (Version 2.2)*, Computer Science Department Australian National University.