Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas - ICE
Departamento de Ciência da Computação – DCC

Algoritmos e Estrutura de Dados
Emparelhamento Máximo

Vitor Freitas e Souza (102350037) – vitorfs@gmail.com
Rodrigo Martins (xxxxx) – drg.martins@gmail.com

JUIZ DE FORA
2013

1 INTRODUÇÃO

Este trabalho teve por objetivo implementar algoritmos para resolver o problema de emparelhamento máximo para grafos quaisquer e para grafos ponderados, utilizando a abordagem do **Khum**. Para o desenvolvimento do algoritmo de Khum foi necessário o estudo e implementação do algoritmo **Húngaro**.

Além dos algoritmos propostos, implementamos uma nova função, com base em nossa reflexão e entendimento do problema para resolver problemas de emparelhamento máximo em grafos não-dirigidos e não-ponderados, com complexidade quadrática.

2 EMPARELHAMENTO MÁXIMO

O problema de emparelhamento máximo consiste em analisar um dado grafo G = (V, E) e extrair um subconjunto de E de arestas duas a duas não adjacentes. Um emparelhamento M é dito **máximo** se não existir um outro subconjunto M' tal que M' > M. Um emparelhamento é dito **maximal** se não existir um emparelhamento M' do qual M faça parte.

Podemos afirmar ainda que um dado grafo G = (V, E) pode ter mais de um emparelhamento maximal. Um emparelhamento **perfeito** é um emparelhamento máximo cujo conjunto de arestas duas a duas é equivalente a V/2, ou seja, satura todos os vértices do grafo, o que quer dizer que somente existe um emparelhamento perfeito se o número de vértices for par.

Em linhas gerais, o problema de emparelhamento máximo consiste em encontrar um emparelhamento máximo num grafo não-dirigido.

3 IMPLEMENTAÇÃO

O desenvolvimento foi feito com a linguagem de programação C utilizando TADs necessárias para solução do problema. A fim de padronizar o código, as funções e os TADs foram descritas em inglês utilizando *snake case* para representar nomes compostos.

O código foi indentado com dois espaços e o uso de chaves para delimitar blocos foi tomado como obrigatório mesmo para blocos de controle de fluxo com um comando apenas. Toda definição relevante foi separada em arquivos separados, com sua implementação C e seu respectivo cabeçalho.

A entrada de dados é feita via argumentos de linha de comando, sendo o primeiro parâmetro o nome do arquivo texto onde a definição do grafo se encontra, seguindo o seguinte padrão: a primeira linha descreve a quantidade de

vértices do grafo, a partir da segunda linha as arestas são representadas por um par de vértices separados por espaços. Admite-se um terceiro valor em cada linha para os casos onde o grafo é ponderado.

3.1 AMBIENTE

Linguagem de programação: C
Compilador: GCC 4.2.1

3.2 TIPOS ABSTRATOS DE DADOS

Devido a linguagem de programação utilizada para a implementação do problema, as TADs foram definidas em **structs**, a fim de adicionar maior semântica ao código, melhorando a implementação, manutenibilidade e entendimento do código.

3.1.1 Graph

Estrutura base da implementação para representar o grafo do problema. A TAD Graph representa uma matriz de adjacência e juntamente em seu cabeçalho definimos as principais funções para trabalhar com este tipo de estrutura. A partir da leitura do arquivo texto de entrada, a aplicação converte os conjuntos de vértices dois a dois, juntamente com seu peso (quando se aplica) e cria uma instância de Graph. É importante salientar que, por ser uma matriz de adjacência, podendo ser de um grafo ponderado, o peso 0 representa a ausência de adjacência.

```
typedef struct {
  int vertex_count;
  int** arcs;
  Vertex* vertex;
} Graph;
```

A estrutura armazena a quantidade de vértices para possibilitar a iteração na matriz, que foi representada por um ponteiro de ponteiro de inteiros denominado *arcs*. Sua alocação é feita de forma dinâmica, permitindo ainda remoção e inserção de vértices em tempo de execução.

A TAD armazena ainda um ponteiro de vértices para auxiliar na solução do problema, descrito pela TAD *Vertex*, que somente faz sentido de existir dentro do contexto de *Graph*, por isso a mesma fora definida dentro do mesmo cabeçalho de *Graph*.

```
typedef struct {
  int vertex;
```

```
  int degree;
  int saturated;
} Vertex;
```

A justificativa de utilizar esta estrutura auxiliar é para que consigamos ter um acesso rápido ao grau dos vértices sem precisar realizar uma operação O(n) e determinar de forma confortável se um dado vértice está saturado ou não. Ambas as operações são realizadas em O(1) e sua manutenção é feita com custo mínimo.

Devido às limitações da linguagem e o paradigma de programação utilizado, não é possível encapsular as funções juntamente com as estruturas, de forma que as funções de *Graph* sejam serviços específicos da mesma. No entanto suas funções foram definidas juntamente com sua estrutura, e evidentemente só faz sentido utilizá-las no contexto de *Graph*.

- **insert_arc**: operação realizada em O(1). Embora admitimos que o grafo seja não-orientado, é feito a redundância de representação na matriz de adjacência, adicionando a aresta nas posições (x, y) e (y, x). O motivo disto é que em problemas específicos dentro da solução do emparelhamento máximo é conveniente ter as duas representações. Quando não necessário, realizamos a busca somente na triangular superior, assumindo *n* como sendo o número de vértices, com a seguinte estratégia:

```
for (i = 0 ; i < n ; i++) {
  for (j = i ; j < n ; j++) {
    //código relevante
  }
}
```

- **remove_arc**: operação simples realizada em O(1). Considerando a redundância de representação das arestas, é feito a remoção de ambas posições (x, y) e (y, x);
- **get_adjacency**: esta função retorna um ponteiro de inteiros com a lista de adjacências de um dado vértice *v*. Operação realizada em O(n).
- **get_vertex_degree**: operação realizada em O(n) para determina o grau de um dado vértice *v*. A utilização desta função é feita em contextos específicos e normalmente dentro de um laço de *n* repetições, sendo *n* o número de vértices do grafo, para preencher os graus dos vértices dentro da estrutura *Vertex*.