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 Ferreira Martins (102350042) – martins.drg@gmail.com

JUIZ DE FORA
2013

SUMÁRIO

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*.

Funções 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*.

- `get_ordered_adj`: função responsável por retornar uma lista de vértices ordenadas em ordem crescente pelo grau do vértice. Esta função faz uso da função `get_vertex_degree`, preenchendo primeiramente a estrutura Vertex com todos os graus de seus respectivos vértices em O(n^2). Em seguida realizamos uma operação de ordenação em O(n log n), utilizando o método *quicksort*. Este método foi implementado para apoiar a implementação da nossa função de emparelhamento máximo, onde era relevante ter uma lista ordenada por graus.

- `read_graph`: função utilizada com intuito de converter a estrutura descrita em um arquivo texto de entrada, no formato descrito no tópico 3 deste documento, para a TAD *Graph*.

3.1.2 Set

Esta estrutura foi definida com o intuito de apoiar a implementação do algoritmo Húngaro e por conseguinte o algoritmo de Khum. Em linhas gerais a TAD Set é basicamente uma representação de conjuntos e a abordagem utilizada foi a de uma lista encadeada com descritor.

```
typedef struct set {
  int vertex;
  struct set* next;
} Set;
```

Com esta estrutura temos somente o número do vértice e um ponteiro para o próximo elemento. Uma instância do tipo Set com valor NULL é dito um conjunto vazio. Como em alguns casos específicos da implementação admite-se conjuntos vazios e precisamos passar estas instâncias por parâmetro, que podem vir a serem alocadas dentro do contexto deste método, este cenário se tornou um empecilho para o desenvolvimento, sendo necessário a implementação de um descritor denominado HeaderSet. Independente deste conjunto estar vazio ou não, dentro do contexto da aplicação ele está sempre alocado, possibilitando assim sua rastreabilidade e passagem como parâmetro sem perder a referencia.

```
typedef struct header_set {
  int nodes;
  Set* first;
} HeaderSet;
```

A estrutura é composta então por um ponteiro para o elemento inicial da lista encadeada e por um inteiro denominado *nodes* que representa a quantidade de elementos contidos na lista. Sua manutenção é feita com custo mínimo, com incrementos e decrementos a partir das funções de inserção e remoção no conjunto.

As operações que normalmente gostaríamos de realizar com estas estruturas basicamente é de armazenar um subconjunto de vértices V' de um dado grafo G = (V, E). Seja de suas partições (nos casos de grafos bipartido) ou de conjuntos de vértices saturados.

Funções de *Set*:

- `bipartite`: esta função recebe uma instância de *Graph*, assumindo que este grafo seja bipartido, realiza uma busca em profundidade recursiva e retorna via argumentos de referencia as duas partições do grafo. Vale salientar que, se o grafo possuir mais de uma componente conexa, pode existir mais de uma representação possível, com diferentes números de vértices em cada partição, uma vez que não é possível determinar qual a partição correta de cada vértice das componentes conexas. O tamanho e a disposição dos conjuntos será influenciada pelo valor do vértice inicial, que começamos a busca em profundidade. Como precisamos separar uma entrada em duas saídas, o retorno é feito via ponteiros.

- `bipartite_define_header_set`: implementação recursiva da busca em profundidade. A operação é feita em O(n^2).

- `builds_neighborhood_header_set`: função responsável por construir um conjunto de vértices *NS* contendo os vizinhos de um dado conjunto *S*. Esta operação é necessária para implementação do caminho aumentante.

- `compare_header_set`: função booleana para determinar a igualdade de dois conjuntos. Retorna 1 se os conjuntos forem iguais e 0 caso contrário. Esta função recebe um conjunto NS e um conjunto T como parâmetro. O conjunto T sempre será subconjunto de NS, então marca-se todos os elementos de T na lista e verifica se NS contém algum elemento que não foi marcado.

- `subtraction_header_set`: esta função recebe dois conjuntos *NS* e *T* como parâmetro e retorna um terceiro conjunto com a diferença de ambos.

3.1.3 Arcs

Esta estrutura possui duas semânticas dentro da aplicação. A primeira delas representa um conjunto recíproco de arestas, ou seja, se existe aresta (*u, v*) também existe aresta (*v, u*). A segunda representa um caminho aumentante, que para percorrê-lo depende-se de um vértice inicial.

```
typedef struct arcs {
```

```
  int n;
  int **arcs;
} Arcs;
```

A definição deste TAD se assemelha com a definição do TAD *Graph*, no entanto possui uma semântica diferente para a aplicação. Com intuito de tornar a implementação mais clara e tirar do programador a responsabilidade de interpretar o contexto em que a estrutura está sendo utilizada para compreender sua semântica, definimos um novo TAD. A matriz de *Graph* representa uma matriz de adjacência. Já a matriz de *Arcs* representada pelo atributo *arcs* por sua vez representa uma matriz de *n * 2*, considerando *n* o número de vértices de um dado grafo. Nesta estrutura armazenamos os vértices da estrutura, seu peso e a informação de que este vértice está saturado ou não.

Funções de *Arcs*:

- `non_saturation_header_set`: função responsável por verificar se um dado conjunto não é M-saturado, retornando assim o valor -1. Caso contrário, a função retorna o primeiro vértice não M-saturado. Esta função recebe uma instância de *Set* e uma instância de *Arcs*, que representa o conjunto dos vértices que já foram saturados. Esta operação é realizada em O(n).

- `saturation_header_set`: função responsável por verificar se um dado conjunto é M-saturado. Operação realizada em O(n).

- `symmetric_difference_arcs`: esta função é responsável por criar a diferença simétrica de um dado conjunto M, representando o conjunto de arestas saturados e um conjunto P, representando um caminho aumentante. Considerando *n* como sendo o tamanho do caminho de P, esta operação é realizada em O(n).

- `augmenting_path`: função responsável por criar um caminho aumentante, partindo de dois vértices não saturados, sendo uma a origem e o outro o destino. Para criar este caminho, utilizamos a matriz de adjacência representada pela estrutura *Graph* e o conjunto de arestas M-saturadas representadas pela estrutura *Arcs*.

3.3 ALGORITMOS

Os algoritmos do problema de emparelhamento máximo foram implementado em arquivos separados, a fim de separar as responsabilidades do código fonte. Suas implementações foram agrupadas em um programa principal com intuito de permitir a execução de todos.

3.3.1 Maximal Matching

Este algoritmo foi implementado com base em nosso entendimento do problema, sem utilizar qualquer ideia contida em artigos relevantes sobre o problema. Este método nem sempre consegue determinar um emparelhamento máximo e/ou perfeito, mas se mostrou eficiente em achar emparelhamentos maximal. O resultado foi satisfatório considerando a complexidade do procedimento, que é feito em O(nˆ2).

A execução do algoritmo é feita a partir de um dado grafo G = (V, E), representado por uma matriz de adjacência. Embora consideramos que o grafo G não possui laços, o método faz um tratamento da estrutura a fim de remover os possíveis laços e garantir a execução correta do algoritmo, independente da consistência do grafo de entrada.

A estratégia de busca pela solução é baseada em um conjunto de vértices em ordem crescente de grau. A execução inicia em um laço invariante de *n* repetições, tomando *n* como o número de vértices do grafo. A partir da lista de vértices ordenada denominada *v*, iniciamos o emparelhamento pelo vértice de menor grau *v1*. Se este vértice não for saturado, realizamos uma busca de seus vértices adjacentes em O(n) e pegamos a primeira adjacência não-saturada *v2* e inserimos esta aresta no *Graph* de solução. Em seguida, marcamos ambos os vértices como saturados e removemos todas as adjacências de v1 e v2 do *Graph.*

A execução continua até que o último vértice do conjunto *v* de vértices ordenados seja analisado. Após o fim do laço invariante, retornamos a matriz de adjacência do emparelhamento.

Esta solução retorna um emparelhamento maximal, que por ventura pode vir a ser um emparelhamento máximo ou perfeito. Conseguimos identificar se o emparelhamento é perfeito, mas no entanto não conseguimos garantir a melhor solução.

3.3.2 Hungarian

3.4 ALGORITMOS E ESTRUTURAS AUXILIARES

Para resolver alguns problemas referentes a implementação dos algoritmos de emparelhamento máximo e das estruturas que apoiaram o desenvolvimento, foi necessário a codificação alguns algoritmos conhecidos de ordenação e de estruturas de pilhas.

3.4.1 Algoritmos de Ordenação