UGRAPH: BIBLIOTECA PARA MANIPULAÇÃO DE GRAFOS

FLÁVIO KEIDI MIYAZAWA – IC-UNICAMP
fkm@ic.unicamp.br www.ic.unicamp.br/~fkm/ugraph

Última atualização: 24 de setembro de 2002

Sumário

1 Introdução

A biblioteca UGRAPH está escrita em C ANSI e é voltada para a manipulação de grafos simples e não orientados. A biblioteca faz uso de outros programas disponíveis na rede. Usa o programa NEATO para a visualização dos grafos e usa algumas rotinas disponibilizadas pelo ZIB-Berlin.

Esta biblioteca está na versão 0 (talvez seja melhor colocar versão -1). Por ser a primeira versão disponibilizada para alunos, peço por favor que sempre que for localizado um erro, me avise no email `fkm@ic.unicamp.br` para que eu possa corrigir e atualizar a página.

Obrigado,

Flávio.

2 Header

Para usar a biblioteca, use o header

```
#include ¨ugraph.h¨
```

3 Sistema Operacional

Esta biblioteca está fácil para ser testada tanto em ambiente Linux como em ambiente Windows.

Para isso, declare as seguintes variáveis globais:

```
extern gr_os_type gr_sistema;
extern gr_outputtype gr_output;
```

Depois, atribua valores para estas variáveis, no início do programa principal, conforme o sistema operacional.

Windows

```
gr_sistema = GR_WINDOWS;
gr_output = GR_GIF_WINDOWS;
```

Linux

```
gr_sistema = GR_LINUX;
gr_output = GR_POSTSCRIPT_LINUX;
```

Em Windows, o programa para visualização dos grafos é realizado através do programa PAINTBRUSH. Em Linux, a visualização é feita através do programa GV. Veja um exemplo de uso no programa `exemplo.c`. Neste arquivo estão listados outras alternativas de visualização.

4 Tipos

`graphtype:` Este é o principal tipo que você deve usar nesta biblioteca. É o tipo do grafo.

`boolean:` Este tipo pode ter um de dois valores: `false` (falso) ou `true` (verdadeiro).

`colortype:` Vértices e arestas podem ter cor. Algumas cores possíveis são: `BLACK, WHITE, RED, GREEN, BLUE`.

5 Rotinas da biblioteca

Dividimos as rotinas desta biblioteca nas seguintes seções: manipulação de grafos, manipulação de vértices, manipulação de arestas, adjacência de arestas, visualização do grafo, algoritmos em grafos e rotinas auxiliares.

A seguir detalhamos cada uma delas:

5.1 Rotinas para manipulação de grafos

`gr_creategraph:` Inicializa um grafo do tipo `graphtype`.

Retorna o ponteiro do grafo criado. *protótipo:* `graphtype * gr_creategraph(int maxvertex,int maxedges)`

resultado: ponteiro do grafo, NULL se não conseguiu criar. `maxvertex:` é o número máximo de vértices que o grafo poderá ter (poderá ter qualquer quantidade de vértices entre 0 e `maxvertex`)

`maxedges:` é o número máximo de arestas que o grafo poderá ter (poderá ter qualquer quantidade de arestas entre 0 e `maxedges`)

A quantidade de memória alocada é proporcional a `maxvertex+maxedges`.

`gr_closegraph:` Libera a memória alocada para o grafo.

protótipo: `void gr_closegraph(graphtype *g)`

`g:` é ponteiro para o grafo.

`gr_setgraphname:` Comando para atribuir um nome ao grafo.

protótipo: `boolean gr_setgraphname(graphtype *g,char *s)`

resultado: `true` se obteve sucesso, `false` caso contrário.

`g:` é ponteiro para o grafo.

`s:` ponteiro para o novo nome do grafo.

`gr_getgraphname:` Comando para obter o nome do grafo.

protótipo: `boolean gr_getgraphname(graphtype *g,char *s)`

resultado: `true` se obteve sucesso, `false` caso contrário.

`g:` é ponteiro para o grafo.

`s:` ponteiro para gravar nome do grafo.

`gr_writegraph:` grava um grafo no formato padrao usado pelo programa (arquivo texto).
Na primeira linha do programa devem vir 2 numeros, digamos N e M que são os números de vértices e de arestas. Depois devem vir N linhas, cada linha contém o nome de um vértice. Depois devem vir M linhas, uma para cada aresta. Cada linha de aresta tem os nomes dos extremos e depois vem o peso da aresta (separados por espaços em branco).
protótipo: `boolean gr_writegraph(graphtype *g,char *filename)`
resultado: `true` se obteve sucesso, `false` caso contrário.
`g:` é ponteiro para o grafo.
`filename:` nome do arquivo de saída.

`gr_readgraph:` lê um grafo no formato padrão usado pelo programa.
protótipo: `boolean gr_readgraph(graphtype *g,char *filename)`
resultado: `true` se obteve sucesso, `false` caso contrário.
`g:` é ponteiro para o grafo.
`filename:` nome do arquivo de entrada.

`gr_cleangraph:` faz uma cópia do grafo, mas com os índices dos vértices e arestas seqüênciais. Útil quando queremos usar os índices das arestas e vértices iguais a $0, 1, \ldots,$, seqüêncialmente (sem índices no meio da seqüência que não estejam usados). Um exemplo onde isso pode ocorrer é na indexação das colunas de um programa linear.
protótipo: `boolean gr_cleangraph(graphtype *dest,graphtype *source)`
resultado: `true` se obteve sucesso, `false` caso contrário.
`source:` ponteiro para o grafo original, o grafo continua existindo depois da operação.
`dest:` ponteiro para o grafo destino.

`gr_getcleangraph:` como a rotina `gr_cleangraph`, mas retorna um ponteiro para o grafo gerado.
protótipo: `graphtype * gr_getcleangraph(graphtype *g)`
resultado: ponteiro para o novo grafo, `NULL` se não teve sucesso.
`g:` ponteiro para o grafo original, o grafo continua existindo depois da operação.

`gr_writegraph_neato:` gravar um grafo no formato usado pelo programa NEATO.
protótipo: `boolean gr_writegraph_neato(graphtype *g,char *filename)`
resultado: `true` se obteve sucesso, `false` caso contrário.
`g:` é ponteiro para o grafo.
`filename:` nome do arquivo de saida.

`gr_number_components:` retorna o número de componentes do grafo
protótipo: `int gr_number_components(graphtype *g)`
resultado: caso tenha problemas para acessar `g`, retorna -1.
`g:` é ponteiro para o grafo.

`gr_number_vertices:` retorna o número de vértices do grafo
protótipo: `int gr_number_vertices(graphtype *g)`
resultado: caso tenha problemas para acessar `g`, retorna -1.
`g:` é ponteiro para o grafo.

`gr_number_edges:` retorna o número de arestas do grafo
protótipo: `int gr_number_arestas(graphtype *g)`
resultado: caso tenha problemas para acessar `g`, retorna -1.
`g:` é ponteiro para o grafo.

5.2 rotinas para manipulação de vértices

Os vertices são armazenadas em uma lista ligada combinada com vetor. Assim, é possivel percorrer os vértices usados através da lista ligada, obtendo o primeiro vértice e o vértice seguinte a outro. Cada vértice tem um índice e

um nome. Dois vértices distintos tem índices distintos. Se nunca for removido um vértice durante a construção de um grafo, então os índices dos vértices do grafo serão $0, 1, 2, \ldots, n-1$, onde n é o número de vértices do grafo.

`gr_getfirstvertex:` retorna o primeiro vértice do grafo.
protótipo: `int gr_getfirstvertex(graphtype *g)`
resultado: retorna o índice do primeiro vértice da lista de vértices.
`g:` é ponteiro para o grafo.

`gr_getnextvertex:` dado um vértice, retorna o próximo vértice do grafo.
protótipo: `int gr_getnextvertex(graphtype *g,int v)`
resultado: retorna o índice do próximo vértice, depois de `v`.
`g:` é ponteiro para o grafo.
`v:` é índice do vértice do qual se quer o próximo.

`gr_getvertexindex:` dado o nome de um vértice, retorna o índice do vértice no grafo.
protótipo: `int gr_getvertexindex(graphtype *g,char *name)`
resultado: retorna o índice do vértice com nome `name`. Retorna -1 se não encontrou.
`g:` é ponteiro para o grafo.
`name:` nome do vértice procurado.

`gr_existsvertex:` verifica a existência de um vértice.
protótipo: `boolean gr_existsvertex(graphtype *g,int v)`
resultado: retorna `true` se existe um vértice com índice `v`, e retorna `false` caso contrário.
`g:` é ponteiro para o grafo.
`v:` é índice do vértice do qual se quer verificar a existência.

`gr_insertvertex:` insere um novo vértice no grafo.
protótipo: `int gr_insertvertex(graphtype *g, char *name)`
resultado: retorna -1 se não conseguiu inserir um novo vértice e retorna o índice do novo vértice caso tenha conseguido inserir novo vértice. Obs.: cuidado, estas rotinas reutilizam índice de vértices anteriormente removidos.
`g` é ponteiro para o grafo.
`name` é o nome do novo vértice, deve necessariamente ser diferente de qualquer outro vértice existente. Posteriormente melhoraremos a versão desta biblioteca para que contemple também vértices sem nome.

`gr_getvertexname:` obtém o nome de um vértice.
protótipo: `boolean gr_getvertexname(graphtype *g,int v,char *name);`
resultado: `true` se obteve o nome, `false` caso contrário.
`g:` é ponteiro para o grafo.
`v:` é o índice do vértice em `g`.
`name:` ponteiro onde será copiado o nome do vértice.

`gr_vertexname:` retorna o ponteiro do nome de um vértice.
protótipo: `char *gr_vertexname(graphtype *g,int v);`
resultado: `true` se obteve o nome, `false` caso contrário.
`g:` é ponteiro para o grafo.
`v:` é o índice do vértice em `g`.

5.3 rotinas para manipulação de arestas

As arestas são armazenadas em uma lista ligada combinada com vetor. É possivel percorrer as arestas usadas através da lista ligada, obtendo a primeira aresta e a aresta seguinte a outra.

As arestas que são incidentes a um vértice também são armazenadas em uma lista ligada (para cada vértice) e também podem ser percorridas de forma análoga.

Cada aresta tem um índice. Duas arestas distintas têm índices distintos. Se nunca for removido uma aresta durante a construção de um grafo, então os índices das arestas no grafo serão $0, 1, 2, \ldots, m-1$, onde m é o número total de arestas no grafo.

Dado um par (u, v) correspondente a uma aresta (onde u e v são os índices dos vértices extremos de uma aresta), é possível recuperar rapidamente seu índice (tempo médio constante).

`gr_insertedge:` insere uma aresta no grafo.
protótipo: `int gr_insertedge(graphtype *g,char *name,int u,int v,double weight)`
resultado: `false` se já existe uma aresta com os extremos
`u` e `v`, ou se ocorrer um caso de inviabilidade; caso contrário retorna `true`.
`g:` é ponteiro para o grafo.
`name:` nome da aresta (por enquanto este dado não está sendo utilizado)
`u,v:` índices dos vértices extremos da aresta.
`weight:` peso da aresta.

`gr_getfirstedge:` retorna a primeira aresta do grafo.
protótipo: `int gr_getfirstedge(graphtype *g)`
resultado: retorna o índice da primeira aresta da lista de arestas.
`g:` é ponteiro para o grafo.

`gr_getnextedge:` dado uma aresta, retorna a próxima aresta do grafo.
protótipo: `int gr_getnextedge(graphtype *g,int e)`
resultado: retorna o índice da próxima aresta da lista de arestas, depois de `e`.
`g:` é ponteiro para o grafo.

`gr_getedge:` dado uma aresta através dos seus extremos, retorna o índice da aresta.
protótipo: `int gr_getedge(graphtype *g,int u, int v)`
resultado: índice da aresta, dado os índices dos vértices extremos da aresta. Se não houver aresta, retorna -1.
`g:` é ponteiro para o grafo.
`u,v:` são índices para os vértices que são extremos da aresta.

`gr_getedgename:` obtém o nome de uma aresta.
protótipo: `boolean gr_getedgename(graphtype *g,int e,char *name)`
resultado: `true` se obteve o nome, `false` caso contrário.
`g:` é ponteiro para o grafo.
`e:` é o índice do aresta em `g`.
`name:` é o nome onde será retornado o nome da aresta.

`gr_edgename:` retorna o ponteiro do nome de uma aresta.
protótipo: `char * gr_edgename(graphtype *g,int e)`
resultado: ponteiro do nome da aresta, se não existir aresta retorna `NULL`.
`g:` é ponteiro para o grafo.
`e:` é o índice do aresta em `g`.

`gr_getedgehead:` obtém um dos extremos de uma aresta (head).
protótipo: `int gr_getedgehead(graphtype *g,int e)`
resultado: índice do vértice que é o primeiro extremo de `e:`.
`g:` é ponteiro para o grafo.
`e:` é o índice da aresta em `g`.

`gr_getedgetail:` obtém um dos extremos de uma aresta (tail).
protótipo: `int gr_getedgehead(graphtype *g,int e)`
resultado: índice do vértice que é o segundo extremo de `e`.
`g:` é ponteiro para o grafo.
`e:` é o índice da aresta em `g`.

`gr_existsedge:` verifica se um índice é válido para uma aresta.
protótipo: `boolean gr_existsedge(graphtype *g,int e)`
resultado: retorna `true` se existe uma aresta com índice `e`, e retorna `false` caso contrário.
`g` é ponteiro para o grafo.
`e` é índice da aresta que se quer verificar a existência.

`gr_getedgeweight:` obtém o peso de uma aresta.
protótipo: `boolean gr_getedgeweight(graphtype *g,int e,double *peso)`
resultado: `true` se obteve o peso da aresta `e`, e `false` caso contrário.
`g:` é ponteiro para o grafo.
`e:` é índice da aresta que se quer verificar a existência,
`peso:` ponteiro para onde deve ser armazenado o peso da aresta.

`gr_edgeweight:` retorna o peso de uma aresta.
protótipo: `double gr_edgeweight(graphtype *g,int e)`
resultado: peso da aresta, da aresta `e`, e `MINIMUMDOUBLE` caso a aresta não exista.
`g:` é ponteiro para o grafo.
`e:` é índice da aresta que se quer verificar a existência.

`gr_setedgeweight:` atribui o peso de uma aresta.
protótipo: `boolean gr_setedgeweight(graphtype *g,int e,double peso)`
resultado: `true` se conseguiu atribuir o peso da aresta `e`, e `false` caso contrário.
`g:` é ponteiro para o grafo.
`e:` é índice da aresta que se quer verificar a existência,
`peso:` novo peso a ser atribuído.

5.4 rotinas para manipulação de arestas adjacentes a um vértice

As arestas que são incidentes a um vértice também são armazenadas como listas ligadas e também podem ser percorridas seqüencialmente.

`gr_getvertexfirstadj:` retorna o índice da primeira aresta incidente a um vértice.
protótipo: `int gr_getvertexfirstadj(graphtype *g,int v)`
resultado: índice da primeira aresta da lista de incidência do vértice `v`. Se o vértice não tem arestas, retorna -1.
`g:` é ponteiro para o grafo.
`v:` é o índice do vértice.

`gr_getvertexnextadj:` dado uma aresta incidente a um vértice, retorna o índice da próxima aresta também incidente ao vértice.
protótipo: `int gr_getvertexnextadj(graphtype *g,int v,int e)`
resultado: índice da próxima aresta da lista de incidência do vértice de índice `v`, depois da aresta `e`. Se não houver próxima aresta incidente, retorna -1.
`g:` é ponteiro para o grafo.
`v:` é o índice do vértice,
`e:` é o índice da aresta incidente a `v`.

5.5 Rotinas para visualização do grafo

A visualização do grafo é realizada através de dois programas: NEATO (para gerar um grafo em arquivo postscript/gif/jpg/...) e GV (para visualizar um grafo gerado em postscript).

O grafo pode ter cores nas arestas e nos vértices. Para ver a lista de cores, veja o tipo `colortype`.

`gr_viewgraph:` visualiza o grafo.

este comando necessita que estejam instalados os programas NEATO e ou o programa GV (Linux), ou GSVIEW (Windows), ou o PAINTBRUSH (Windows).

O programa NEATO é obrigatório e pode ser obtido para Linux e Windows no link: `http://www.research.att.com/sw/tools/graphviz/download.html`. No site está disponibilizado o programa `neato.exe` para plataforma Windows. Obs.: As vezes o programa NEATO gera um desenho diferente para um mesmo grafo.

O programa GV é para visualizar o grafo através de saida postscript. Para Windows, você pode usar no lugar do `gv`, o programa `gsview` (veja os comentários/código na rotina `gr_viewgraph` do arquivo `ugraph.c` para saber como mudar). O `gsview` pode ser obtido no link `http://www.cs.wisc.edu/~ghost/gsview/`.

Outra alternativa é não usar um visualizador de arquivos postscript. Mas alguma ferramenta que visualize arquivos .gif e .jpg. Um que é disponível em Windows é o programa PAINTBRUSH.

protótipo: `void gr_viewgraph(graphtype *g)` `g:` é ponteiro para o grafo.

`gr_setvertexcolor:` pinta um vértice de uma cor.

protótipo: `boolean gr_setvertexcolor(graphtype *g,int v,colortype cor)`
resultado: `true` se conseguiu pintar o vértice e `false` caso contrário.
`g:` é ponteiro para o grafo.
`v:` é o índice do vértice a pintar,
`cor:` é a cor.

`gr_setedgecolor:` pinta uma aresta de uma cor.

protótipo: `boolean gr_setedgecolor(graphtype *g,int e,colortype cor)`
resultado: `true` se conseguiu pintar a aresta e retorna `false` caso contrário.
`g:` é ponteiro para o grafo.
`e:` é o índice da aresta a pintar,
`cor:` é a cor.

`gr_getedgecolor:` obtém a cor de uma aresta.

protótipo: `boolean gr_getedgecolor(graphtype *g,int e,colortype *cor)`
resultado: `true` se conseguiu obter a cor da aresta e retorna `false` caso contrário.
`g:` é ponteiro para o grafo.
`e:` é o índice da aresta a pintar,
`cor:` é o ponteiro para o local onde deve gravar a cor.

`gr_paintgraphedges:` pinta todas as arestas do grafo.

protótipo: `void gr_paintgraphedges(graphtype *g,colortype cor)`
`g:` é ponteiro para o grafo.
`cor:` é a cor.

`gr_paintvectoredges:` pinta com uma cor as arestas do grafo, dadas através de um vetor.

protótipo: `void gr_paintvectoredges(graphtype *g,int *edges, int nedges, colortype cor)`
`g:` é ponteiro para o grafo.
`edges:` ponteiro para o vetor dos índices das arestas.
`nedges:` número de arestas no vetor.
`cor:` é a cor.

`gr_paintgraphvertex:` pinta todos os vértices do grafo.

protótipo: `void gr_paintgraphvertex(graphtype *g,colortype cor)`
`g:` é ponteiro para o grafo.
`cor:` é a cor.

`gr_paintgraph:` pinta todos os vértices e arestas do grafo com uma cor.

protótipo: `void gr_paintgraph(graphtype *g,colortype cor)`

`g:` é ponteiro para o grafo.

`cor:` é a cor.

`gr_paintgraphscale:` pinta todas as arestas do grafo, com uma escala simples, dependendo do peso. As arestas do grafo com peso entre maior que 2/3 são pintadas com `blue`, as arestas com peso em (1/3,2/3] são pintadas com `red` e as arestas com peso menor ou igual a 1/3 são pintadas de `black`. a ser melhorado no futuro.

protótipo: `void gr_paintgraphscale(graphtype *g)`

`g:` é ponteiro para o grafo.

`gr_paintsubgraph:` pinta as arestas de um subgrafo de um grafo

dado um grafo `g` e um subgrafo `sub` (o subgrafo foi obtido do grafo) a rotina localiza as arestas através dos nomes do vértices. O grafo `g` é inicialmente pintado de preto e as cores do subgrafo `sub` são copiadas para o grafo `g`

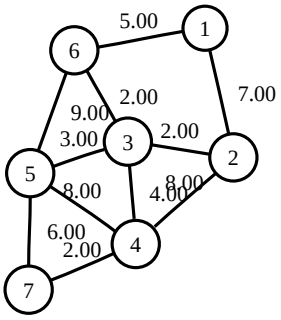
protótipo: `void gr_paintsubgraph(graphtype *g,graphtype *sub)`

`g, sub:` é o ponteiro do grafo e do subgrafo, respectivamente.

5.6 Rotinas para encontrar subgrafos e outras estruturas

Algumas rotinas para encontrar certas estruturas já foram desenvolvidas, como algoritmo para encontrar uma árvore geradora de peso mínimo, corte de peso mínimo separando dois vértices, árvore de cortes de Gomory-Hu, etc.

Vamos exemplificar com o seguinte grafo (os desenhos dos grafos abaixo foram obtidos executando o programa exemplo.c):



Grafo com 7 vertices e 11 arestas

`gr_minimumspanningtree:` obtém uma árvore geradora de peso mínimo.

protótipo: `boolean gr_minimumspanningtree(graphtype *gr,graphtype *mst)`

resultado: `true` se conseguiu gerar a árvore geradora de peso mínimo e `false` caso contrário.

`g:` é o ponteiro para o grafo original

`mst:` é o ponteiro para onde vai ser gerado o subgrafo.

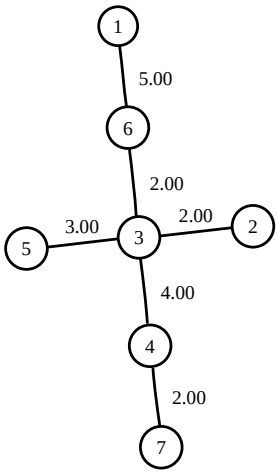
`gr_getminimumspanningtree:` obtém uma árvore geradora de peso mínimo.

protótipo: `boolean gr_getminimumspanningtree(graphtype *gr,int *mstedges, int *mstnedges)`

resultado: `true` se conseguiu gerar a árvore geradora de peso mínimo e `false` caso contrário.

`mstedges:` é o ponteiro para um vetor onde os indices das arestas da árvore será armazenado

`mstnedges:` é o ponteiro para o local onde será armazenado o número de arestas da árvore.



Arvore geradora de G, com 7 vertices

`gr_min_st_cut:` obtém um grafo que é o corte de peso mínimo separando dois vértices.

protótipo: `boolean gr_min_st_cut(graphtype *g,int s_g,int t_g, graphtype *stcut,double *cutweight)`

resultado: `true` se conseguiu gerar o corte mínimo que separa o vértice `s_g` e o vértice `t_g`. Caso contrário retorna `false`.

`g:` é o ponteiro para o grafo original

`stcut:` é o ponteiro para onde vai ser gerado o subgrafo que forma o corte mínimo.

`cutweight:` ponteiro para o local onde será gravado o peso do corte.

`gr_get_min_st_cut:` obtém um grafo que é o corte de peso mínimo separando dois vértices.

protótipo: `boolean gr_get_min_st_cut(graphtype *g,int s_g,int t_g, int *cutedges, int *ncutedges ,double *cutweight)`

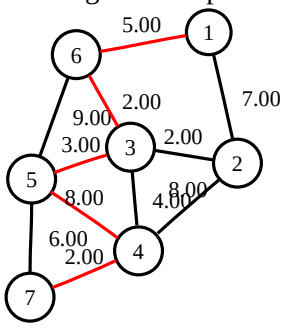
resultado: `true` se conseguiu gerar o corte mínimo que separa o vértice `s_g` e o vértice `t_g`. Caso contrário retorna `false`.

`g:` é o ponteiro para o grafo original

`cutedges:` é o ponteiro para um vetor onde serão armazenados os índices das arestas do corte.

`ncutedges:` é o ponteiro para o local onde será armazenado a quantidade de arestas do corte.

`cutweight:` ponteiro para o local onde será gravado o peso do corte.



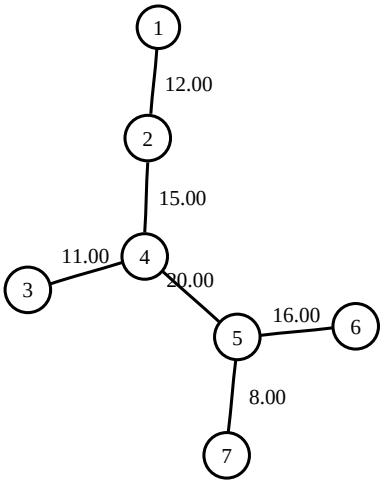
Valor do corte separando vertices 4 e 5 = 20.00000

`gr_generate_ghc_tree:` gera a árvore de cortes de Gomory-Hu.

protótipo: `void gr_generate_ghc_tree(graphtype *g,graphtype *tree)`

`g:` é o ponteiro para o grafo original

`tree:` ponteiro para onde vai ser gerado a árvore de cortes de Gomory-Hu



Arvore de Gomory Hu

`gr_find_min_edge_ghc_tree:` dado uma árvore de corte de Gomory-Hu e dois vértices desta árvore, retorna o índice da aresta que representa o menor corte
protótipo: `boolean gr_find_min_edge_ghc_tree(graphtype *tree,int s,int t,int *edgeinde`
resultado: `true` se conseguiu gerar encontrar a aresta de peso mínimo que separa `s` e o vértice `t`. Caso contrário retorna `false`.
`tree:` é o ponteiro para o grafo original
`s, t:` índices dos vértices da árvore para o qual queremos encontrar a aresta do corte mínimo.
`edgeindex:` ponteiro para o local onde será gravado o índice da aresta correspondente ao corte.

5.7 Outras rotinas úteis

Algumas rotinas de uso geral úteis durante a programação.

`gr_getnextname:` lê a próxima palavra *(token)* de um arquivo. As palavras são separadas por caracteres "branco"(são considerados caracteres branco: espaços, tabs, nova linha, fim de linha).
protótipo: `void gr_getnextname(char *str,FILE *fp);`
A próxima palavra é retornada em `str`.

Índice Remissivo