Projeto Euler1736

Guia de uso

Índice

O QUE É O PROJETO EULER1736?

O projeto Euler1736 é uma aplicação CLI (Command Line Interface) feita para facilitar o manuseio de grafos digitalmente. Feita como parte da disciplina de Estruturas de Dados II no ano de 2019 na FCT - UNESP de Presidente Prudente. O nome foi escolhido para homenagear o ano e o autor do primeiro artigo científico a estudar os grafos, Leonhard Euler.

O Euler1736 foi desenvolvido usando a linguagem C e o Sistema Operacional Ubuntu. Testes serão feitos para garantir a maior compatibilidade possível entre os diversos sistemas operacionais mais usados, mas não há garantias.

QUEM FEZ O EULER1736?

Quem fez o Euler1736 foi o Rafael, durante o 2º ano do curso de Ciência da Computação da FCT - UNESP.

Ciência da Computação - FCT UNESP

Rafael Araújo Chinaglia

2º ano

COLABORANDO COM O EULER1736

O Euler1736 tem um repositório no GitHub para que todos possam contribuir e construir um projeto melhor e maior! O link é github.com/chinaglia-rafa/Grafos.

TABELA DE FUNCIONALIDADES

O Euler1736 tem as seguintes funcionalidades implementadas:

#01	**Interface CLI**	A CLI do projeto Euler1736 conta com comandos como `help`, `status`, `select` e `debug` para ajudar a controlar diversos grafos ao mesmo tempo, bem como a forma de exibição das informações e etc.
#02	**Carregamento de múltiplos grafos**	Os grafos podem ser carregados pelos arquivos presentes na pasta /grafos, seguindo o padrão de arquivo que será apresentado nos capítulos a seguir através dos comando `load`.
#03	**Tabela de adjacência**	A aplicação permite acesso à tabela de adjacência de todos os grafos carregados através do comando `adj`.
#04	**Verificação de propriedades**	Grafos podem ser analisados (usando busca de profundidade, algoritmo de *Kruskal* e demais informações do grafo) para constatar propriedades como conectividade, tipo de grafo, presença de ciclos e etc através do comando `props`.
#05	**Busca de profundidade**	Grafos podem ser analisados através da busca de profundidade e a tabela resultante pode ser exibida e/ou salva em um arquivo localizado em /saved através do comando `dfs`.
#06	**Busca de largura**	Grafos podem ser analisados através da busca de largura e a tabela resultante pode ser exibida e/ou salva em um arquivo localizado em /saved através do comando `bfs`.
#07	**Caminho mínimo**	Grafos podem ser analisados através de Dijkstra ou Bellman-Ford e as tabelas resultantes podem ser salvas na pasta /saved através do comando `path`.
#08	**Modo de edição**	Modo de edição com capacidade de adicionar e remover vértices (*vertexes*) e arestas (*edges*), bem como salvar os grafos alterados em arquivos na pasta /grafos através do comando `edit`.

FORMATO PADRÃO PARA ARQUIVOS DE GRAFOS

Para representarmos nossos grafos em arquivos, usaremos o seguinte padrão, considerando cada linha como uma linha do arquivo:

```
1            [tipo de grafo: 0 = grafo, 1 = dígrafo]
5            [quantidade de vértices]
0 3 2        [representação da aresta que liga 0 a 3 com peso 2]
0 2 5        [representação da aresta que liga 0 a 2 com peso 5]
3 4 1
4 0 2                          ...
2 1 2
1 0 1        [representação da aresta que liga 1 a 0 com peso 1]
```

Dessa forma, conseguimos representar grafos, dígrafos e todas as suas arestas e vértices componentes.

<u>Observação</u>: como pode ser deduzido, a aplicação extrai os vértices com base nas arestas, o que significa que vértices que não possuem nenhuma ligação não terão *apelidos* como os outros têm, e ao invés disso serão gerados ordenada e sistematicamente com base no vértice de maior índice + 1.

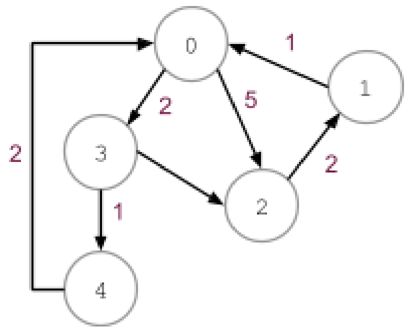
EXEMPLO BÁSICO DE USO

Para ilustramos o funcionamento do projeto, vamos usar o grafo grafos/exemplo.grafo como base e usar alguns comandos. Lembre-se, a qualquer momento você pode usar o comando `help` para ver uma lista de comandos possíveis.

1. ARQUIVO `exemplo.grafo`

O arquivo exemplo.grafo tem os seguintes conteúdos:

```
1
5
0 3 2
0 2 5
3 4 1
4 0 2
2 1 2
1 0 1
```



2. CARREGANDO O GRAFO

Para carregar o grafo, usaremos o comando `load`, da seguinte maneira:

```
load exemplo.grafo
```

3. VERIFICAR GRAFOS CARREGADOS

A qualquer momento, use o comando `status` para visualizar a situação atual da aplicação.

```
status
```

A saída será algo do tipo:

```
Grafos carregados:
* [1] digrafo carregado de "grafos/g1.grafo" (5 vértices)

Use o comando select <id> para selecionar outro grafo como ativo.

Status do DEBUG: on
Log atual é "logs/20191130.log”
```

O relatório de status nos informa que o grafo principal é o 1 através do * marcado antes de sua linha, bem como o status do DEBUG e o caminho do arquivo de log atual. Para ligar e desligar o debug em tela, use o comando `debug off` ou `debug on`.

Para facilitar o trabalho com nosso exemplo, vamos também ligar o modo KEEP, que não apaga o console a cada comando:

```
keep on
```

4. VISUALIZANDO O GRAFO

Vamos agora usar o comando `adj` para visualizar a árvore de adjacência do nosso dígrafo.

```
adj
```

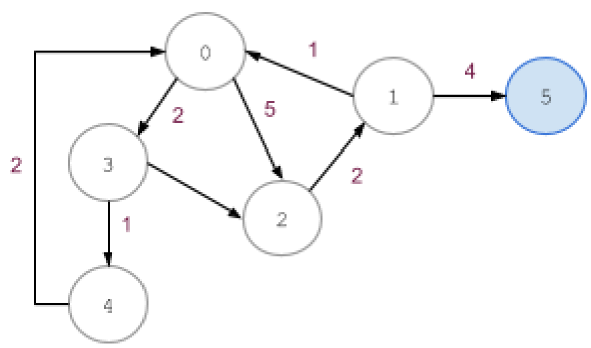
A saída será algo no formato:

```
         0   3   2   4   1
       ---------------
  0 |   0   2   5   0   0
  3 |   0   0   0   1   0
  2 |   0   0   0   0   2
  4 |   2   0   0   0   0
  1 |   1   0   0   0   0
```

Perceba que os números nas bordas são nossos vértices, enquanto os números destacados nos cruzamentos são os pesos das arestas, como por exemplo (4, 0) com peso 2 e (0, 2) com peso 5.

5. ATUALIZANDO O GRAFO

Que tal agora adicionar um novo vértice 5 para conseguirmos o dígrafo abaixo?



Para isso, usaremos o modo de edição com o comando `edit` para **a)** criar o vértice 5 e **b)** criar a aresta (1, 5) com peso 4.

```
edit
add --vertex 5
add --edge 1 5 4
quit --save
```

Lembre-se de usar a flag `--save`, caso contrário você sairá e as alterações não serão salvas no arquivo grafos/exemplo.grafo. Para verificar se existem alterações não salvas, use o comando `status` e verifique os grafos com a observação:

```
[1] dígrafo carregado de "grafos/g1.grafo" (6 vértices) [Alterações não salvas]
```

6. FAZER UMA BUSCA

Para fazer uma busca por profundidade (*Deep First Search*), use o comando `dfs <raiz>` e o <id> do grafo que deseja analisar. Como o nosso dígrafo já é o principal, não precisamos informar seu <id>. Vamos começar a busca pelo vértice 0.

```
dfs 0
```

O comando acima imprimirá uma tabela contendo os tempos de descoberta e finalização de cada vértice do grafo, marcando com um * a raiz.

```
    Raiz é o elemento 0
    ___________________________
   | vértice | cor |  d  |  f  |
   |_________|_____|_____|_____|
 * |    0    |  p  |   1 |  12 |
   |    3    |  p  |   2 |   5 |
   |    2    |  p  |   6 |  11 |
   |    4    |  p  |   3 |   4 |
   |    1    |  p  |   7 |  10 |
   |    5    |  p  |   8 |   9 |
```

7. PROPRIEDADES DO GRAFO

Para ver uma série de propriedades do grafo, use o comando props para ver que nosso dígrafo é

a. um dígrafo
b. conexo
c. cíclico
d. não árvore
e. não floresta.

ESTRUTURAS PRINCIPAIS

Além das diversas estruturas auxiliares usadas na aplicação, existem estruturas principais que compõem o cerne do Euler1736, cujas explicações seguem abaixo.

struct Grafo

```
struct Grafo {
    int size;
    int type;
    int item[100][100];
    int index[100];
    char filename[200];
    short is_from_file;
    short edited;
};
```

Essa estrutura é usada para armazenar cada grafo carregado. Os itens `size`, `type`, `item (arestas)` e `index (cada apelido de vértice)` são usados para montar a tabela de adjacência do grafo, enquanto os demais campos são para controle de edição e origem do grafo carregado.

struct Linha_profundidade

```
struct Linha_profundidade {
    int index;
    int d;
    int f;
    enum_color color;
};
```

A estrutura Linha_profundidade representa cada linha da tabela gerada pelo comando `dfs`. `index` se refere ao índice do vértice referenciado, `d` é o tempo de descobrimento, `f` é o tempo de finalização e `color` pode ser `white`, `grey` ou `black`, para representar o status de cada vértice durante a execução.

struct Linha_largura

```
struct Linha_largura {
    int index;
    int parent;
    int d;
    enum_color color;
};
```

A estrutura Linha_largura representa cada linha da tabela gerada pelo comando `bfs`. `index` se refere ao índice do vértice referenciado, `d` é a distância da raiz, `parent` é o pai do vértice e `color` pode ser `white`, `grey` ou `black`, para representar o status de cada vértice durante a execução.

Filas, pilhas e árvores

Durante a execução de algoritmos, estruturas como structs de tabelas, que são combinações de structs de linha com structs de tabela com vetores delas para representar as árvores resultantes, foram utilizadas, bem como pilhas, filas e etc.

Para as seguinte funcionalidades, as estruturas de dados discriminadas foram utilizadas (funções presentes no arquivo `grafos.h`)

Algoritmo	Função Principal	Fila	Recursão	Pilha	Struct de tabela
Matriz de adjacência	`load_grafo_from_file`	-	-	-	-
Busca por profundidade	`busca_profundidade`	-	X	X	X
Busca por largura	`busca_largura`	X	-	-	X
Dijkstra	`path_dijkstra`	X	-	-	X
Bellman-Ford	`path_bellford`	-	-	-	X
Kruskal	`kruskal`	X	-	-	X

IMPLEMENTAÇÕES PRINCIPAIS

Busca por profundidade

Vamos agora analisar o funcionamento do algoritmo de busca por profundidade. Para localizarmos os trechos de código, usaremos a seguinte notação:

`(arquivo, função, linha inicial)`

1. Inicialização

o primeiro passo é criar a estrutura de dados necessária na função principal `busca_profundidade()` `(grafos.h, busca_profundidade, 663)`, ela sendo **`struct`** `Tabela_profundidade`, e inicializar seus valores iniciais conforme a função `tbl_prof_reset()` `(grafos.h, tbl_prof_reset, 355)`, usando o `-1` como nossa representação para NULL.

2. Visitando vértices

Com a tabela inicializada, e para cada vértice, visitaremos o vértice caso seja branco. Para isso, usaremos a função `tbl_prof_visit()` `(grafos.h, tbl_prof_visit, 399)` que é recursiva.

3. No vértice

```
399 void tbl_prof_visit(struct Tabela_profundidade* t, int i, struct Grafo m) {
400     t->linha[i].color = grey;
401     dfs_tempo += 1;
402     t->linha[i].d = dfs_tempo;
403     for (int j = 0; j < m.size; j++) {
404         if (m.item[i][j] == 0) continue;
405         if (t->linha[j].color == white) {
406             tbl_prof_visit(t, j, m);
407         }
408     }
409     t->linha[i].color = black;
410     dfs_tempo += 1;
411     t->linha[i].f = dfs_tempo;
412 }
```

fonte: `(grafos.h, tbl_prof_visit, 399)`, sintetizado.

Tendo visitado um dado vértice `t`, marcaremos sua cor como cinza e seu tempo de descoberta como `dfs_tempo`, que começou em 0. Em seguida, verificaremos quais vértices adjacentes a `t` possuem a coloração branca e os visitaremos usando a mesma função (*l. 406*), recursivamente.

4. Condição de parada

Conforme um dado vértice `t` termina de chamar recursivamente a função `tbl_prof_visit()` `(grafos.h, tbl_prof_visit, 399)` para todos os vértices adjacentes a ele, o vértice `t` ganha um tempo de finalização e a cor preta, indicando que ele não será visitado mais. Quando todos os vértices tiverem sido visitados e possuírem a cor preta, o algoritmo terminará.

No final da execução, todos os vértices possuírão um tempo de descoberta, um tempo de finalização e uma cor relacionada a ele.

Busca por largura

Para o algoritmo de busca por largura, usaremos um *approach* iterativo.

1. Inicialização

Criaremos antes de mais nada uma **`struct`** `Tabela_largura` e uma **`struct`** `Queue` para auxiliar no controle do algoritmo. Em seguida, a função `tbl_larg_reset()` `(grafos.h, tbl_larg_reset, 388)` inicializará a tabela com os valores iniciais, considerando sempre o `-1` como NULL para o pai de cada vértice e como ∞ para a distância da raiz.

2. Visitando os vértices

Para visitar os vértices do grafo, o algoritmo usa o seguinte trecho:

```
while (current_index != -1) {
    for (int i = 0; i < m.size; i++) {
        if (m.item[current_index][i] == 0) continue;
        if (tabela.linha[i].color == white) {
            tabela.linha[i].color = grey;
            tabela.linha[i].d = tabela.linha[current_index].d + 1;
            tabela.linha[i].parent = indexAt(current_index, m);
            push(&queue, i);
        }
    }
    tabela.linha[current_index].color = black;
    current_index = pop(&queue);
}
```

`fonte: (grafos.h, busca_largura, 605), sintetizado.`

usando como arestas os cruzamentos encontrados na tabela de adjacência armazenada em `m.item[i][j]`.

O algoritmo anota cada vértice adjacente a um dado vértice `t`, branco e que pode ser visitado em uma fila, e repete esse processo para cada vértice que puder retirar da fila.

Para cada vértice adjacente a `t`, o algoritmo o pinta de cinza, anota sua distância da raiz e seu pai. Quando se esgotam todos os vértices adjacentes a um dado vértice `t`, o algoritmo dá à `t` a cor preta e desenfileira o primeiro elemento da fila, repetindo o processo a partir dele.

3. Condição de parada

O algoritmo é encerrado quando todos os elementos da fila são visitados e removidos dela. Na tabela final, podemos observar que os vértices desconexos possuirão distância da raiz `-1`, indicando que o grafo não é conexo.

Caminho mínimo usando Dijsktra

Para encontrar uma árvore de caminho mínimo usando Dijkstra, usaremos a lógica a seguir.

1. Inicialização

A inicialização é feita usando a função `tbl_dijkstra_reset()` `(grafos.h, tbl_dijkstra_reset, 524)`.

2. Busca por arestas com peso negativo

Uma característica importante do algoritmo de Dijkstra é que ele não pode ser aplicado a grafos que possuam pesos negativos. Por isso, o seguinte trecho de código fica responsável por identificar esse fator e impedir o algoritmo de ser executado.

```
529 for (int i = 0; i < m.size; i++) {
530     for (int j = 0; j < m.size; j++) {
531         if (m.item[i][j] < 0) {
532             printf("[ERRO]: Aresta com peso negativo encontrada!");
533             struct Tabela_dijkstra null_struct;
534             null_struct.size = -1;
535             return null_struct;
536         }
537     }
538 }
```

`fonte: (grafos.h, path_dijkstra, 529), sintetizado.`

No primeiro sinal de uma aresta com peso < 0 (*l. 531*), o algoritmo é interrompido e um sinal é enviado para o programa principal que alerta o usuário.

3. Relaxando as arestas

Dijkstra usa uma função de relaxamento de arestas, que anota uma distância mais eficiente do que a melhor distância conhecida entre a raiz e um dado vértice.

```
int current_vertex = -1;
while (queue_count > 0) {
    current_vertex = get_dijkstra_min(tabela);
    if (current_vertex == -1) break;
    tabela.linha[current_vertex].on_queue = 0;
    //  Relaxa as arestas entre current_vertex e seus adjacentes
    for (int i = 0; i < tabela.size; i++) {
        if (m.item[current_vertex][i] == 0) continue;
        //  Relaxando (current_vertex --> i)
        if (tabela.linha[i].d == -1 || tabela.linha[i].d >
            tabela.linha[current_vertex].d + m.item[current_vertex][i]) {
                tabela.linha[i].d = tabela.linha[current_vertex].d +
                                    m.item[current_vertex][i];
            tabela.linha[i].parent = current_vertex;
        }
    }
}
```

fonte: (grafos.h, path_dijkstra, 529), sintetizado.

O relaxamento acontece enquanto houver elementos na lista. Um a um eles são removidos da fila, sempre optando pelo vértice com distância menor, e a função de relaxamento é executada sobre cada aresta que liga este vértice aos seus adjacentes.

4. Condição de parada

Quando todas os vértices saíram da fila, ou seja, quando todos as arestas foram analisadas para encontrar os melhores caminhos da raiz até cada vértice, o algoritmo pára, e a tabela resultante é impressa contendo a representação de uma árvore parcial de caminho mínimo.

Caminho mínimo usando Bellman-Ford

Para encontrar uma árvore de caminho mínimo usando Bellman-Ford, usaremos a lógica a seguir.

1. Inicialização

Após a criação das tabelas e estruturas necessárias, o algoritmo inicializa a tabela usando -1 no campo de pai para representar NULL e no campo de distância para representar ∞. E por fim marca a distância da raiz como 0.

2. Relaxando vértices

Diferente de Dijkstra, Bellman-Ford vai relaxar todos os vértices $n - 1$ vezes, onde n é o número de vértices do grafo. O relaxamento feito a partir de (*l. 486)* atualiza o melhor caminho para cada vértice a partir de um raiz previamente definida quando ele é mais eficiente do que o melhor conhecido.

```
481 for (int c = 0; c < tabela.size - 1; c++) {
482     for (int i = 0; i < tabela.size; i++) {
483         for (int j = 0; j < tabela.size; j++) {
484             if (m.item[i][j] == 0) continue;
485             // Relaxando (i, j)
486             if (tabela.linha[j].d == -1 ||
487                 tabela.linha[j].d > tabela.linha[i].d + m.item[i][j]) {
488                     tabela.linha[j].d = tabela.linha[i].d + m.item[i][j];
489                     tabela.linha[j].parent = i;
490             }
491         }
492     }
493 }
```

fonte: (grafos.h, path_bellford, 450), sintetizado.

3. Buscando ciclos negativos

Para determinar seu valor de retorno, o algoritmo busca por ciclos negativos através do trecho de código abaixo:

```
497 for (int i = 0; i < tabela.size; i++) {
498     for (int j = 0; j < tabela.size; j++) {
499         if (m.item[i][j] == 0) continue;
500         if (tabela.linha[j].d > tabela.linha[i].d + m.item[i][j]) {
501             struct Tabela_bellford null_struct;
502             null_struct.size = -2;
503             return null_struct;
504         }
505     }
506 }
```

fonte: (grafos.h, path_bellford, 450), sintetizado.

Caso um seja encontrado, o algoritmo retorna FALSE (*l. 503*). Caso contrário, o algoritmo chega ao final e retorna TRUE. Além disso, o algoritmo também imprime uma tabela que permite a montagem de uma árvore parcial de tamanho mínimo.

Detectar ciclos usando Kruskal

Um dos recursos mais importantes do Euler1736 é a detecção de ciclos, usada no relatório do comando `bfs`. Para detectar ciclos, usaremos o algoritmo de Kruskal, que agrupa os vértices em “famílias” através da adição ordenada de arestas ao grafo.

1. Inicialização

Para este algoritmo, usaremos algumas estruturas interessantes. Entre elas, usaremos **`struct`** `Tabela_kruskal` para anotar as *famílias* das árvores geradas até o momento, **`struct`** `static_list arestas[99]` será usada para ordenar as arestas de forma crescente e uma **`struct`** `edge` para cada item da lista ordenada. Em seguida, as estruturas são inicializadas e as arestas são adicionadas em ordem crescente à lista estática.

```
for (int i = 0; i < m.size; i++) {
    for (int j = 0; j < m.size; j++) {
        if (m.item[i][j] == 0) continue;
        struct edge e;
        e.origin = i;
        e.destiny = j;
        e.w = m.item[i][j];
        //  Insere na lista estática
        if (list_root == -1) {
            list_root = 0;
            arestas[list_size].edge = e;
            list_size++;
        } else {
            int current = list_root;
            while (arestas[current].edge.w < e.w &&
                   arestas[current].next != -1)
                current = arestas[current].next;
            if (e.w < arestas[current].edge.w) {
                arestas[list_size].edge = e;
                arestas[list_size].next = current;
                if (current == list_root) list_root = list_size;
                list_size++;
            } else {
                arestas[list_size].edge = e;
                arestas[list_size].next = arestas[current].next;
                arestas[current].next = list_size;
                list_size++;
            }
        }
    }
}
```

fonte: (grafos.h, kruskal, 869), sintetizado.

2. Adicionando arestas

Com a lista de arestas ordenadas por peso, o algoritmo passa a adicionar as arestas uma a uma, agrupando os vértices interligados em árvores e dando a eles a mesma família, representada pelo menor valor de vértice presente na árvore, até que todos os vértices possuam a mesma família, implicando em um grafo conexo, ou todas as arestas sejam processadas.

```
916 int current = list_root;
917     while (current != -1) {
918         //  Detecta ciclo
919         if (tabela.linha[arestas[current].edge.origin].family ==
920             tabela.linha[arestas[current].edge.destiny].family) {
921             tabela.has_ciclo = 1;
922             current = arestas[current].next;
923             continue;
924         }
925         int lowest_family = min(
926             tabela.linha[arestas[current].edge.origin].family,
927             tabela.linha[arestas[current].edge.destiny].family
928         );
929         for (int c = 0; c < m.size; c++) {
930             if (tabela.linha[c].family ==
    tabela.linha[arestas[current].edge.origin].family || tabela.linha[c].family
    == tabela.linha[arestas[current].edge.destiny].family)
931                 tabela.linha[c].family = lowest_family;
932         }
933         current = arestas[current].next;
934     }
```

fonte: (grafos.h, kruskal, 869), sintetizado.

O complexo trecho acima detecta possíveis ciclos que seriam criados e os evita, mas não sem antes anotar na propriedade `has_ciclo` (*l. 921*) da tabela de Kruskal que um ciclo foi encontrado.

MAIS EXEMPLOS NO EULER1736

Selecionar um grafo e fazer uma busca de profundidade

```
load g1.grafo
load g2.grafo
select 2
dfs
```

Verificar se um grafo é árvore

```
load g1.grafo
props
```

Remover o vértice 3 do grafo

```
load g1.grafo
edit
rm --vertex 3
```

Comparar duas tabelas de adjacência

```
load g1.grafo
load g2.grafo
keep on
adj 1
adj 2
```

Criar uma cópia de `g1.grafo` que seja um dígrafo

```
load g1.grafo
edit
type digrafo
saveas g1d.grafo
quit
```

Encontrando uma árvore de caminho mínimo com raiz 4 e salvando ele no arquivo `dijkstra.tbl`

```
load g1.grafo
path 4 dijkstra --save
```

Criação dos vértices 1 e 2 ligados por uma aresta de peso 8 e desconexos do grafo

```
load g1.grafo
edit
add -v 1
add -v 2
add -e 1 2 8
quit --save
```