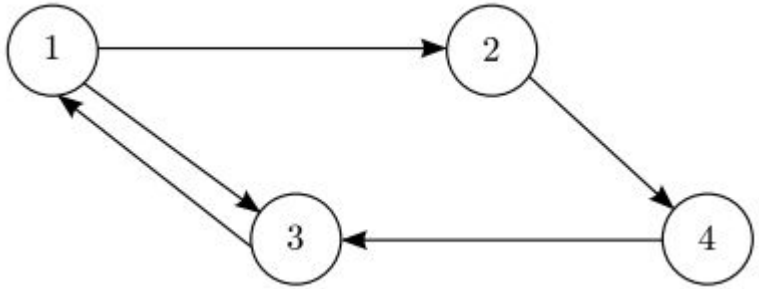
Resolução do problema EP01

Geraldo Rodrigues de Melo Neto
Gustavo Duarte Ventino
Maria Luisa Gabriel Domingues
Pedro de Araújo Ribeiro
Lucas Marques Pinho Tiago

O Problema:

Dadas as conexões entre sites na internet, devemos calcular a média de cliques necessários para ir de um site para todos os outros.

Exemplo:



1: para 2 e 3 um clique, para 4 dois cliques, totalizando 4

2: para 4 um clique, 3 dois cliques, 1 três cliques, totalizando 6

3: para 1 um clique, 2 dois cliques, 4 três cliques, totalizando 6

4: para 3 um clique, 1 dois cliques, 2 três cliques, totalizando 6

Temos um total de 22 cliques para os 4 sites, como cada site vai para 3 outros, a média será 22/(4 x 3) = 22/12

Entradas:

A entrada consiste em diversos conjuntos de dados seguindo o modelo:

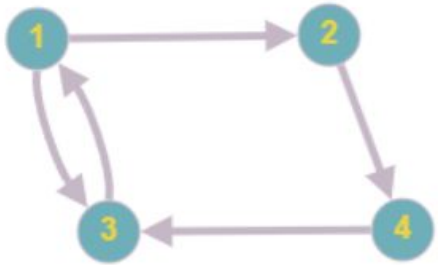
- Uma linha com n pares de números A B, representado uma conexão do site A para o B, terminado por um 0 0 que indica o fim do conjunto
- Outro 0 0 para encerrar a entrada de dados

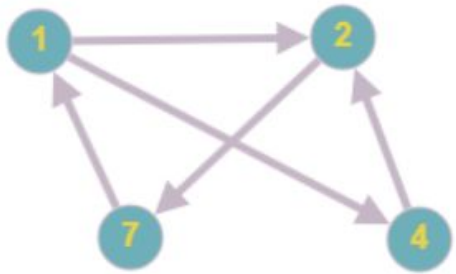
Exemplo:

1 2 2 4 1 3 3 1 4 3 0 0

1 2 1 4 4 2 2 7 7 1 0 0

0 0



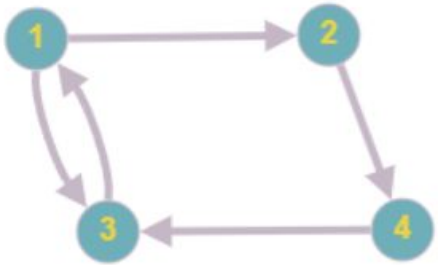


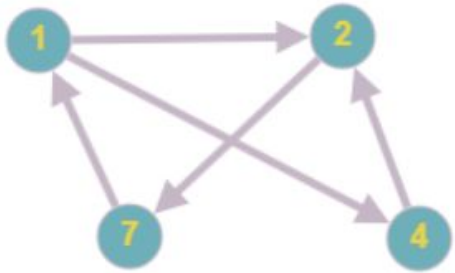
Saídas:

A saída será a média de cliques para cada conjunto no seguinte formato:

Case 1: average length between pages = 1.833 clicks

Case 2: average length between pages = 1.750 clicks





Nossa Resolução:

Decidimos resolver esse problema com Grafos pois podemos representar cada site como vértices de um grafo e suas conexões como arestas direcionadas entre os vértices.

Definimos dois TADs nos quais estão, respectivamente, classes presentes: uma para o grafo e outra para os vértices do grafo.

Nossa Resolução:

- Graph.hpp : contém, de atributos, um array com os vértices do grafo, além do número de vértices.
  - De métodos temos um método para a geração do grafo, um método de solução e impressão da solução;
- Vertex.hpp: contém, de atributos, o id do vértice e sua posição no vetor.
  - De métodos temos apenas print, que imprime o vértice dado.

Graph.hpp

```cpp
#include "Vertex.hpp"
#include <fstream>
#include <queue>
class Graph
{
public:
  std::vector<Vertex*> vertices;
  int size;

//Construtores:
  Graph();
  Graph(int size);

//Faz toda a leitura de entrada e cria o grafo:
  static Graph* readGraph(int a, int b);

//Verifica se um id está no grafo
  int inGraph(int id);

  Vertex* getVet(int id);

//Imprime todo o grafo:
  void print();

  float solve();

  int bfs(Vertex* v);
};
```

Vertex.hpp

```cpp
#include <vector>
#include <string>
#include <iostream>

class Vertex{
public:
  int id;
  int pos;
  std::vector<Vertex*> adjacency;

  Vertex();
  Vertex(int id, int pos);

  //add do vertice na adjacencia
  std::vector<Vertex*> getAdjacency();
  //Adiciona o vertice v na lista de adjacencia
  void addToAdjacency(Vertex *v);
  //Printa as informacoes de id e valor do vertice atual
  void print();
  //Printa a lista de adjacencia do vertice atual
  void printAdjacency();
};
```

Método para resolução do problema

Para resolver o problema, primeiro fizemos a leitura e construção do grafo na função read graph e depois invocamos a função solve. É importante acrescentar que, devido a formatação do input, o ID do vértice nem sempre será correspondente a sua posição na lista de adjacências então fazemos um tratamento para acomodar isso.

A função solve consiste da aplicação de um BFS em todo vértice que durante sua execução conta a quantidade mínima de cliques necessárias para se chegar aos demais vértices e contabiliza esses caminhos, ao final o total é dividido pela quantidade n de vértices vezes n-1.

Leitura de dados:

main.cpp > ...

```cpp
int main() {
  int a, b, cnt = 0;
  std::queue<float> queue;
  std::cin >> a;
  std::cin >> b;
  while(a != 0 && b != 0){
    Graph* g = Graph::readGraph(a,b);
    queue.push(g->solve());
    std::cin >> a;
    std::cin >> b;
  }
```

Graph.cpp > ...

```cpp
Graph * Graph::readGraph(int a, int b) {
  int temp1, temp2, n = 0, m = 0, counter = 2;
  Vertex *v1, *v2;
  Graph *g;
  g = new Graph();
  v1 = new Vertex(a, 0);
  v2 = new Vertex(b, 1);
  v1->addToAdjacency(v2);
  g->vertices.push_back(v1);
  g->vertices.push_back(v2);
  std::cin >> m;
  std::cin >> n;
```

Graph.cpp > ...

```cpp
  while(m != 0 && n != 0){
    temp1 = g->inGraph(m);
    if(temp1==-1){
      v1 = new Vertex(m, counter);
      g->vertices.push_back(v1);
      temp1 = counter;
      counter += 1;
    }
    else
      v1 = g->vertices.at(temp1);
    temp2 = g->inGraph(n);
    if(temp2==-1){
      v2 = new Vertex(n, counter);
      g->vertices.push_back(v2);
      temp2 = counter;
      counter += 1;
    }
    else
      v2 = g->vertices.at(temp2);
    g->vertices.at(temp1)->addToAdjacency(g->vertices.at(temp2));
    std::cin >> m;
    std::cin >> n;
  }
  g->size = g->vertices.size();
  return g;
}
```

Resolução:

Graph.cpp > ...

```
83  float Graph::solve(){
84    float sum = 0;
85    for(auto i : vertices)
86      {
87        sum += bfs(i);
88      }
89    return (sum/(size*(size-1)));
90  }
91
92  int Graph::bfs(Vertex* v){
93      bool visited[size];
94      int distances[size];
95      int sum = 0;
96      for(int i = 0; i<size; i++){
97        visited[i] = false;
98        distances[i] = 0;
99      }
```

Graph.cpp > ...

```
101     std::queue<int> q;
102     int u;
103     visited[v->pos] = true;
104     q.push(v->pos);
105     while (!q.empty())
106     {
107         u = q.front();
108         Vertex* v1 = getVet(u);
109         q.pop();
110         for (auto i : v1->getAdjacency())
111         {
112             if (!visited[i->pos])
113             {
114               //distancia do filho é a do pai + 1
115                 distances[i->pos] = distances[u]+1;
116                 visited[i->pos] = true;
117                 q.push(i->pos);
118             }
119         }
120     }
121
122     for(int i = 0; i<size; i++)
123       sum += distances[i];
124
125     return sum;
126 }
```

FIM

Referências:

Repositório com os códigos fonte: https://replit.com/@PedroRibeiroA12/S09EP01

Site usado para ilustrar os grafos: https://graphonline.ru/en/