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:

Há um jogo onde se transformar uma palavra em outra trocando apenas um caractere de cada vez, devemos criar um método para contar quantas transformações são necessárias para ir de uma palavra **X** para uma palavra **Y**, dado um dicionário fixo.

Exemplo:

spice => slice=> slick => stick => stock

spice => stock = 4 transformações

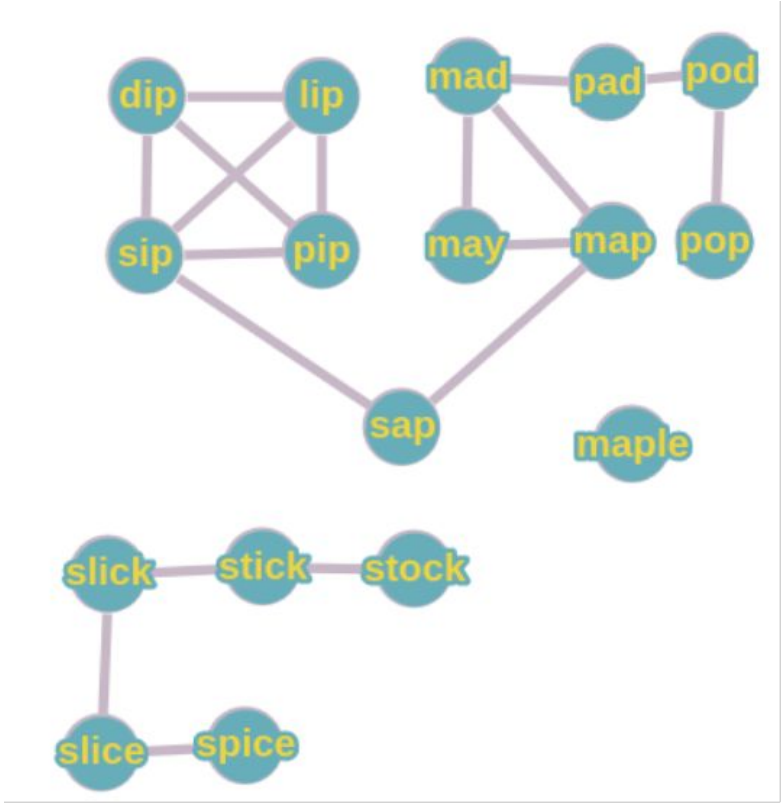
Entradas:

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

- Número de testes **n**;
- Uma sessão de até **200 linhas com o dicionário**, sendo uma palavra por linha;
- Um * para **encerrar o dicionário**;
- Um par de palavras por linha representando o estado inicial e final das transformações, respectivamente;
- É obrigatório que os pares de palavras dados representem transformações possíveis.

Entradas:

```
1
dip
lip
mad
map
maple
may
pad
pip
pod
pop
sap
sip
slice
slick
spice
stick
stock
*
spice stock
may pod
```

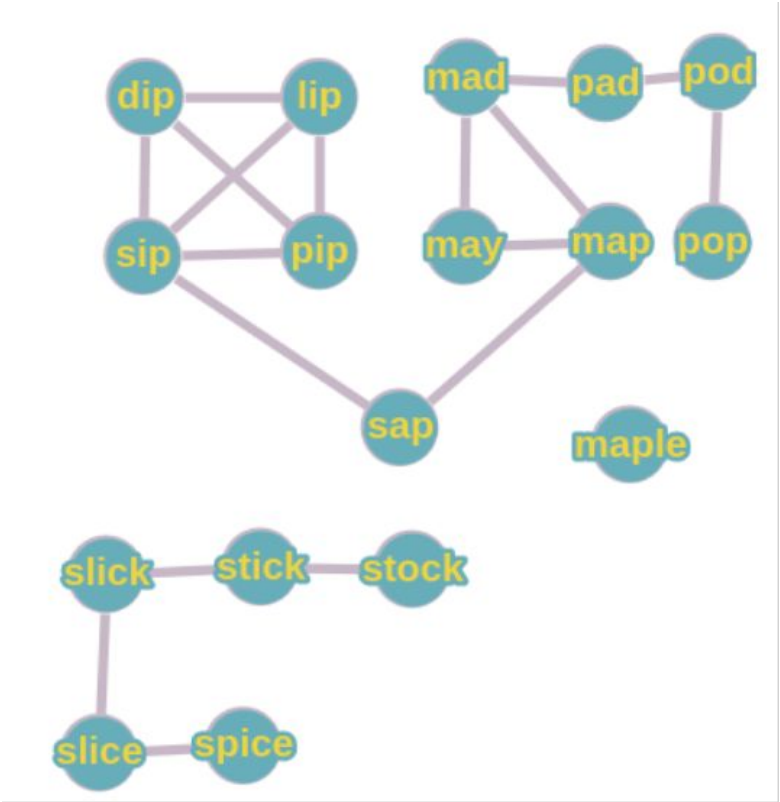


Saídas:

A saída será o par de palavras seguido da quantidade mínima de transformações, um par por linha com os conjuntos de dados separados por um espaço em branco.

Saídas:

spice stock 4
may pod 3



Nossa Resolução:

Decidimos resolver esse problema com Grafos pois podemos representar cada palavra como vértices de um grafo e suas transformações imediatas como arestas 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;
  - Métodos para verificações de strings e inserções baseadas nessas verificações.
- Vertex.hpp: contém, de atributos, um par de id de vértices (gerado após input) e o tamanho da aresta.
  - De métodos temos apenas print, que imprime o vértice dado.

```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 *connect(std::vector<std::string> dictionary);
Vertex* exists(std::string str);
Vertex* addVertex(std::string str, int *id);
//soluçaõ do problema;
int solve(std::string start, std::string end);

//Imprime todo o grafo:
  void print();
  void resetMarks() ;
//Imprime a resposta
  void printResul();
};
```

```cpp
#include <vector>
#include <string>
class Vertex
{
public:
  //Atributos
  std::vector<Vertex*> adj;
  std::string value;
  int id;
  bool marked {false};

  //Construtores:
  Vertex();
  Vertex(int id, std::string value);

  //Printa as informacoes de id
  //e valor do vertice atual
  void print();
};

```

Método para resolução do problema

Para resolver o problema, primeiro fizemos a leitura do dicionário na main e depois o transformamos em um grafo a partir da função *connect*, em seguida para cada par de palavras aplicamos a função solve e computamos seu resultado. Repetimos o processo para cada conjunto de dados.

A função solve consiste da aplicação de um BFS que durante sua execução conta a quantidade mínima de transformação necessárias para se chegar à palavra destino partindo da palavra de partida.

Leitura de dados:

```cpp
for (int i = 0; i < nroCasos; i++) {
  std::vector<std::string> dictionary;

  // captura dos termos do dicionario
  for (int i = 0; i < 200; i++) {
    std::cin >> str;
    if (str == "*")
      break;
    dictionary.push_back(str);
  }

  // montando o grafo a partir do dicionario
  Graph *g = Graph::connect(dictionary);
```

```cpp
// conecta palavras que sao diferentes por uma letra de diferenca:
Graph *Graph::connect(std::vector<std::string> dictionary) {
  Graph *g = new Graph();
  int id = 0;
  Vertex *a, *b;

  for (int i = 0; i < dictionary.size(); i++) {
    b = new Vertex(id, dictionary[i]);
    id++;
    g->vertices.push_back(b);
  }

  for (int i = 0; i < dictionary.size(); i++) {
    a = g->exists(dictionary[i]);
    for (int j = 0; j < dictionary.size(); j++) {
      b = g->exists(dictionary[j]);
      if (verify(dictionary[i], dictionary[j]) == 1 && i != j)
        a->adj.push_back(b);
    }
  }

  return g;
}
```

Resolução:

```
31      // buscando resposta:
32      while (true) {
33        int space = 0;
34        std::getline(std::cin, str);
35
36        if (str == "\n")
37          break;
38
39        for (int j = 0; j < str.size(); j++) {
40          if (str[j] == ' ')
41            space = j;
42        }
43
44        if (std::cin.peek() == '\n')
45          break;
46
47        std::cin >> startWord;
48        std::cin >> endWord;
49        std::pair<std::string, std::string> novoCaso =
50            std::make_pair(startWord, endWord);
51        casos.push(novoCaso);
52        queue.push(g->solve(startWord, endWord));
53        g->resetMarks();
54        print[i] ++;
55      }
```

```
56  // Resolução do problema
57  int Graph::solve(std::string start, std::string end) {
58    std::queue<Vertex *> queue;
59    int counter[vertices.size()];
60
61    Vertex *u = exists(start);
62    Vertex *v = exists(end);
63    Vertex *w = NULL;
64
65    queue.push(u);
66    u->marked = true;
67    counter[u->id] = 0;
68
69    while (!queue.empty()) {
70      w = queue.front();
71      queue.pop();
72
73      for (auto i : w->adj) {
74        if (!i->marked) {
75          counter[i->id] = counter[w->id] + 1;
76          i->marked = true;
77          if (i->value == v->value)
78            return counter[i->id];
79          queue.push(i);
80        }
81      }
82    }
83    return 0;
84  }
```

FIM

Referências:

Repositório com os códigos fonte: https://replit.com/@Ventinos/S07EP01

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