**MINISTÉRIO DA EDUCAÇÃO**
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA
CURSO DE ENGENHARIA DE TELECOMUNICAÇÕES – CÂMPUS SÃO JOSÉ

# RELATÓRIO TÉCNICO

## ALGORITMO DE ROTEAMENTO

*Arthur Cadore Matuella Barcella*
*Gabriel Luiz Espindola Pedro*
*Matheus Pires Salazar*

**RESUMO**

Para a realização desta tarefa, o projeto foi orquestrado por partes, primeiramente recebendo o arquivo de configuração e separando suas rotas em uma estrutura de dados que poderia ser consultada diversas vezes ao longo da execução do código.
Após concluir essa etapa inicial, partimos para o desenvolvimento da interface inicial para o usuário final, que mostraria o menu onde poderíamos informar a cidade de destino, para o cálculo das rotas a fim de chegar a cidade de destino.
Assim que finalizamos esse item, encaminhamos as informações para o algoritmo, e então após o devido processamento, o mesmo nos entrega uma lista para consulta das rotas até o destino.
Após finalizarmos essa etapa, solicitamos a cidade de origem para o cliente para escolher o melhor caminho a ser seguido entre a origem e o destino, e também a distância total a ser percorrida, conforme calculado no algoritmo implementado.

14/05/2022

## INTRODUÇÃO

É apresentado o cenário de descobrir a melhor rota em um mapa, onde é necessário que um usuário indique por teclado as cidades de destino e origem para que a partir dessas informações o mesmo tenha um retorno na tela indicando as cidades na sequência que deve ele passar e a distância total a ser percorrida.

## DESENVOLVIMENTO

Na implementação do projeto, as seguintes estruturas de dados foram utilizadas:

- **Tabela Hash (com lista de estrutura):** A tabela hash foi utilizada em nosso projeto para representar o mapa das rotas, esta estrutura é a base para o funcionamento do algoritmo, visto que ele consultará “n” vezes a tabela a fim de determinar os vizinhos da cidade alvo, e o custo de deslocamento entre esses vizinhos e a cidade destino, através da cidade alvo.

  Em nosso projeto, essa estrutura utiliza a chave sendo a cidade alvo em cada ciclo do loop de varredura, e o valor da chave na tabela, é uma lista de estruturas, onde cada estrutura (descrita abaixo), é na verdade, uma rota da tabela.

  Abaixo temos uma captura da estrutura descrita em nosso projeto, como comentado acima, a mesma armazena os valores das rotas e os adiciona em uma lista de estruturas, e então essa lista é adicionada em uma tabela hash, onde a chave do valor na tabela, é a cidade de origem na rota:

```
// Adiciona cidade vizinha à lista de vizinhos da cidade A.
if (map.find(city_a) == map.end()) {
  // Caso não haja cidade A na hashtable de vizinhos, inicializa.
  list<Neighbor> new_list = {Neighbor{city_b, distance}};

  map[city_a] = new_list;
} else {
  // Caso haja adiciona a cidade B à lista de vizinhos da cidade A.
  map[city_a].push_back({Neighbor{city_b, distance}});
}
```

A seção do código acima (condicional if/else) é utilizada para verificar se a chave atual já existe na tabela, caso não exista, é criada uma nova chave e adicionado à fila com o valor, caso exista, é apenas executado "**push_back**" na lista adicionando a estrutura.

- **Lista de strings:** Esta estrutura de dados é utilizada para receber os nomes de cidades do arquivo .csv e então criar uma lista com um nome de cada cidade, no algoritmo apresentado na página da disciplina, essa lista é denominada "lista Q", e é utilizada no projeto para determinar quais cidades são necessárias ainda verificar no mapa (tabela hash). Abaixo segue uma captura da sessão do código que realiza esta operação:

```
void dijkstra(unordered_map<string, list<Neighbor>>& map, list<string>& cities,
              string& destination,
              unordered_map<string, Table_Line>& distance_table) {
  // Passo 1: Cria a tabela distance_table, de forma que,
  // para cada cidade do mapa: distance_table[cidade] = (MAX_INT, next_city)
  // Observação: D[destination] = (0,destination)
  for (auto city : cities) {
    distance_table[city] = Table_Line{"", INT_MAX};
  }
  distance_table[destination] = Table_Line{"", 0};

  // Passo 2: Cria Q, que é uma lista contendo todos os nodos (incluindo nodoA)
  list<string> Q = cities;
```

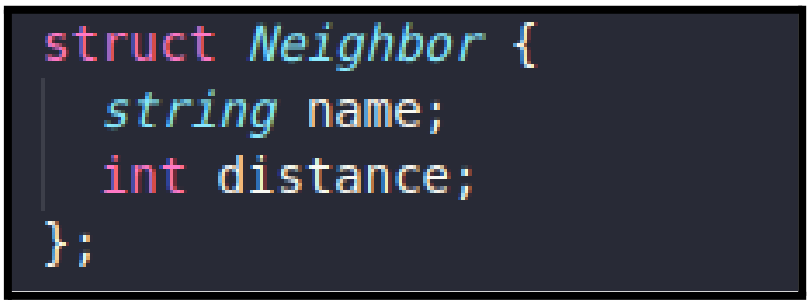
- **Lista de estrutura:**

Por fim, utilizamos uma lista para armazenar as estruturas geradas para cada rota, a estrutura de ***"neighbor"*** recebe no total 2 (dois) parâmetros, sendo eles:

- "***name***": Variável do tipo string que armazena a próxima cidade a ser visitada por um dado vizinho que busca a sua melhor rota.

- **"*distance*"**: Variável do tipo inteira que armazena o valor do "custo", da aresta correspondente no gráfo. Essa informação é utilizada em mapas reais para representar a distância entre os pontos conectados através dela, por exemplo.

**Estrutura "neighbor" descrita acima:**



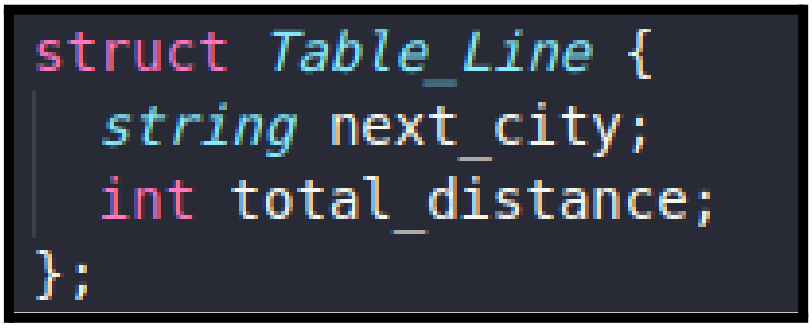
```
struct Neighbor {
  string name;
  int distance;
};
```

Utilizamos também uma segunda estrutura de dados para receber os valores processados no algoritmo a estrutura "**table_line**", que contém também 2 (dois) parâmetros, sendo eles:

- "**next_city**": Variável do tipo string que armazena a próxima cidade a visitar a partir da cidade alvo (cidade que está sendo analisada no ciclo atual do loop).

- "**total_distance**": Variável do tipo inteira que armazena a distância total da cidade analisada (cidade que está sendo analisada no ciclo atual do loop), com a cidade de destino. Ou seja, caso hajam diversas cidades entre a atual e a cidade de destino, esse valor será a somatória das distâncias até a cidade de destino, conforme prevê o protocolo.

**Estrutura "table_line" descrita acima:**



```
struct Table_Line {
  string next_city;
  int total_distance;
};
```

Abaixo também é possível verificar o setor do programa que realiza a verificação da tabela "**distance_table**" e então atualiza caso haja uma rota melhor a ser considerada (a comparação é feita através do custo de cada rota).

```
// Passo 3.2: Para cada nodo v vizinho de u:
//   Calcule a distância do nodo v: dist_v = dist_u + distancia(u,v)
//   Se dist_v for menor do que a distância contida em D[v], faça isto:
//     Atualize D[v] = (dist_v, u)
for (auto neighbor : map[u]) {
  string v = neighbor.name;
  int dist_v = min_distance + neighbor.distance;
  if (dist_v < distance_table[v].total_distance) {
    distance_table[v] = Table_Line{u, dist_v};
  }
}
```

## RESULTADOS

Para comprovar o funcionamento do programa, utilizamos um arquivo de teste para configurar o sistema, e então adicionamos várias rotas já pré-configuradas pelo professor orientador, o arquivo de testes possui rotas simétricas entre os pontos conectados, ou seja, para a operação correta do programa é necessário que seja encaminhado para ele um arquivo com rotas simétricas.

*OBS: O termo rota simétrica significa que o cada rota informada no CSV, precisa possuir uma rota correspondente inversa também informada no arquivo (lembrando que o custo dessa rota não deve ser diferente entre na simetria, apenas os parâmetros de destino e origem são alterados, os valores de distância permanecem iguais), por exemplo:*

Para a rota de Palhoça para são josé informada no arquivo com essa sintaxe:

- **Palhoça, São José, 10**

É necessário que haja a seguinte rota (simétrica a anterior):

- **São José, Palhoça, 10**

A seguir segue uma tabela de exemplo com os dados utilizados no arquivo .csv para teste:

| TABELA DE CSV COM CORRESPONDENTE SIMÉTRICO (EXEMPLO USADO NOS TESTES DO ALGORITMO) | | | | | |
|---|---|---|---|---|---|
| **CIDADE A** | **CIDADE B** | **DISTÂNCIA** | **CIDADE A** | **CIDADE B** | **DISTÂNCIA** |
| Florianópolis | São José | 19 | São José | Florianópolis | 19 |
| Florianópolis | Biguaçu | 24 | Biguaçu | Florianópolis | 24 |
| São José | Biguaçu | 20 | Biguaçu | São José | 20 |
| São José | Palhoça | 6 | Palhoça | São José | 6 |
| São José | São Pedro de Alcântara | 22 | São Pedro de Alcântara | São José | 22 |
| Biguaçu | Antônio Carlos | 17 | Antônio Carlos | Biguaçu | 17 |
| São Pedro de Alcântara | Antônio Carlos | 10 | Antônio Carlos | São Pedro de Alcântara | 10 |
| São Pedro de Alcântara | Angelina | 28 | Angelina | São Pedro de Alcântara | 28 |
| Palhoça | Santo Amaro da Imperatriz | 16 | Santo Amaro da Imperatriz | Palhoça | 16 |
| Palhoça | Paulo Lopes | 40 | Paulo Lopes | Palhoça | 40 |
| Santo Amaro da Imperatriz | Rancho Queimado | 31 | Rancho Queimado | Santo Amaro da Imperatriz | |
| Angelina | Rancho Queimado | 15 | Rancho Queimado | Angelina | 15 |
| Paulo Lopes | Garopaba | 17 | Garopaba | Paulo Lopes | 17 |
| Santo Amaro da Imperatriz | São Bonifácio | 47 | São Bonifácio | Santo Amaro da Imperatriz | 47 |
| São Bonifácio | Garopaba | 118 | Garopaba | São Bonifácio | 118 |

Após compilar e executar o programa inicialmente, o arquivo de configuração é copiado para o tabela hash, onde esta estrutura irá ser tratada como o mapa das rotas contidas no arquivo. Esses dados são armazenados dentro de uma lista de estruturas, onde cada

estrutura contém os valores de cada vizinho, a chave da tabela que contém essa lista, será a cidade que está sendo inspecionada. Abaixo, segue a implementação do loop que importa os dados do arquivo para dentro da lista e portanto para a tabela:

```
// Para cada linha joga informações para map e cities.
while (getline(csv, line)) {
  if (!line.empty()) {
    stringstream ss(line);
    string city_a;
    string city_b;
    int distance;

    // Lê cidade A, cidade B e distância, e insere em suas respectivas
    // variáveis.
    getline(ss, city_a, ',');
    getline(ss, city_b, ',');
    ss >> distance;

    // Adiciona cidade à lista de cidades.
    cities.push_back(city_a);

    // Adiciona cidade vizinha à lista de vizinhos da cidade A.
    if (map.find(city_a) == map.end()) {
      // Caso não haja cidade A na hashtable de vizinhos, inicializa.
      list<Neighbor> new_list = {Neighbor{city_b, distance}};

      map[city_a] = new_list;
    } else {
      // Caso haja adiciona a cidade B à lista de vizinhos da cidade A.
      map[city_a].push_back({Neighbor{city_b, distance}});
    }
  }
}
```

Em seguida, o programa entra em loop permanente, onde solicita ao usuário informar as cidades de origem e destino, para então dar início a execução do algoritmo.

Ao inserir a cidade de destino, que será utilizada como ponto de partida para execução do algoritmo:

```
----------------------Algoritmo de Dijkstra---------------------

Digite o nome da cidade de destino:
```

O programa realiza uma inspeção na tabela hash descrita acima, para verificar se a mesma possui uma chave correspondente ao valor inserido, caso o valor esteja inválido, irá solicitar novamente a mesma informação.

As informações a serem inseridas precisam possuir exatamente a mesma string que a contida no arquivo .csv, caso estejam diferentes o programa não conseguirá realizar a validação corretamente, abaixo podemos verificar uma série de tentativas de inserção inválidas, em que o programa solicitou novamente a informação:

```
----------------------Algoritmo de Dijkstra---------------------

Digite o nome da cidade de destino: florianópolis
Digite o nome da cidade de destino: teste
Digite o nome da cidade de destino: sao josé
Digite o nome da cidade de destino: Sao jose
Digite o nome da cidade de destino: Florianópolis
Digite o nome da cidade de origem: teste
Digite o nome da cidade de origem: florianópolis
Digite o nome da cidade de origem: são josé
Digite o nome da cidade de origem: São José

Menor caminho para chegar em Florianópolis partindo de São José:
```

Após coletar as informações de origem e destino, a aplicação vai armazená-los nas variáveis “**destination**” e “**starting_point**”, e então encaminhá-las para o algoritmo, abaixo podemos verificar a seção do código correspondente:

```cpp
// Variáveis de input de usuário.
string destination;
string starting_point;
while (true) {
  // Pede ao usuário que insira uma cidade do mapa como destino.
  do {
    cout << "Digite o nome da cidade de destino: ";
    getline(cin, destination);
  } while (map.find(destination) == map.end());

  // Tabela de distâncias e caminhos utilizadas pelo algoritmo.
  unordered_map<string, Table_Line> distance_table;
  dijkstra(map, cities, destination, distance_table);

  // Pede ao usuário que insira uma cidade do mapa como origem.
  do {
    cout << "Digite o nome da cidade de origem: ";
    getline(cin, starting_point);
  } while (map.find(starting_point) == map.end());

  // Exibe ao usuário o menor caminho entre origem e destino e informa
  // distância.
  cout << endl;
  cout << "Menor caminho para chegar em " << destination << " partindo de "
       << starting_point << ": " << endl;
```

Em seguida, o algoritmo realiza a verificação da melhor rota possível entre o ponto de origem e o ponto de destino, e então altera em cada ciclo do loop a tabela hash "**distance_table**" que contém as distâncias de cada cidade do mapa para o destino (equivalente a tabela de destinos explicada em aula na apresentação do projeto). Abaixo segue a sessão do código que realiza essa implementação:

```
void dijkstra(unordered_map<string, list<Neighbor>>& map, list<string>& cities,
              string& destination,
              unordered_map<string, Table_Line>& distance_table) {
  // Passo 1: Cria a tabela distance_table, de forma que,
  // para cada cidade do mapa: distance_table[cidade] = (MAX_INT, next_city)
  // Observação: D[destination] = (0,destination)
  for (auto city : cities) {
    distance_table[city] = Table_Line{"", INT_MAX};
  }
  distance_table[destination] = Table_Line{"", 0};

  // Passo 2: Cria Q, que é uma lista contendo todos os nodos (incluindo nodoA)
  list<string> Q = cities;
```

Após conter a tabela "**distance_table**" completa, é realizada uma interação da mesma para determinar o caminho de cidades a ser seguido para chegar no destino que se pede, essa seção do código entra em loop validando se a próxima cidade do caminho é a cidade de destino solicitada, caso não seja, é verificado novamente a tabela contendo esta com sendo a cidade inicial, abaixo segue a sessão do código que realiza essa implementação, para melhor entendimento da operação:

```
string current_city = starting_point;
// Cria string de caminho que será exibida ao usuário
string path = "";
// Itera sobre a tabela de distâncias e caminhos para criar a string
// caminho.
while (current_city != "") {
  path += current_city + " -> ";

  current_city = distance_table[current_city].next_city;
}
```

Finalmente, após a string "path" estar completa, a mesma é impressa no terminal com a sequência de cidades a serem visitadas, abaixo segue a impressão das strings no terminal:

```
----------------------Algoritmo de Dijkstra----------------------

Digite o nome da cidade de destino: Florianópolis
Digite o nome da cidade de origem: São José

Menor caminho para chegar em Florianópolis partindo de São José:
São José -> Florianópolis
Distância total: 19km
```

O código também faz a impressão da distância total do caminho a ser seguido, enfrentando, para determinar esse valor, não é necessário o loop de soma, visto que a tabela "**distance_table**" já contém a distância total para a cidade de destino, abaixo podemos verificar a sessão do código que realiza essa função:

```
cout << path << endl;
cout << "Distância total: " << distance_table[starting_point].total_distance
    << "km" << endl;
```

**MANUAL**

Para utilizar o programa, é necessário realizar os seguintes comandos iniciais, e conter também no diretório, o arquivo de rotas (.csv) que o programa irá utilizar para montar o mapa interno, abaixo segue um descritivo de cada item:

1. **Arquivo de configuração (CSV):**

Para execução do projeto é necessário que haja um arquivo de configuração prévio com os seguintes itens separados pelo caractere vírgula (","): OBJOBJ

- Cidade de origem (string);
- Cidade de destino (string);
- Custo do enlace (inteiro);

Esses parâmetros devem ser informados também de maneira simétrica, onde cada rota terá sua correspondente com as cidades de origem e de destino invertidas, porém mantendo o valor de distância o mesmo (conforme descrito anteriormente no relatório).

- **Demais orientações sobre arquivo:**

É necessário também que na formatação do arquivo, não haja espaços sobressalentes (antes ou após o separador, ou no início da linha), no arquivo também não deve haver linhas sobressalentes no final da tabela.

Deve existir um cabeçalho no arquivo de rotas (.csv), visto que o programa irá desconsiderar a primeira linha do arquivo, caso o cabeçalho não tenha sido implementado no início do arquivo, o programa não irá calcular as rotas corretamente.

O nome do arquivo obrigatoriamente deve ser "**cities.csv**", visto que o programa foi implementado para abrir especificamente esse arquivo no diretório raiz da aplicação.

2. **Comandos iniciais:**

1º Comando: Gera o Makefile para compilação de todos os arquivos do projeto.

**Sintaxe: *cmake CMakeLists.txt***

2º Comando: Compila os arquivos informados do projeto gerando os arquivos para execução.

**Sintaxe: *make***

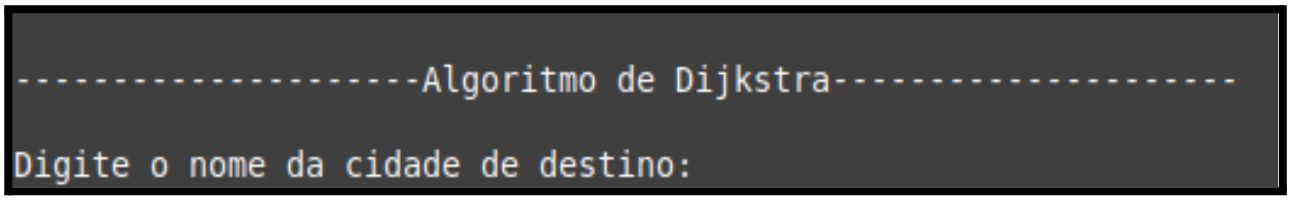
3º Comando: Executar o código do projeto:

**Sintaxe: *make run***

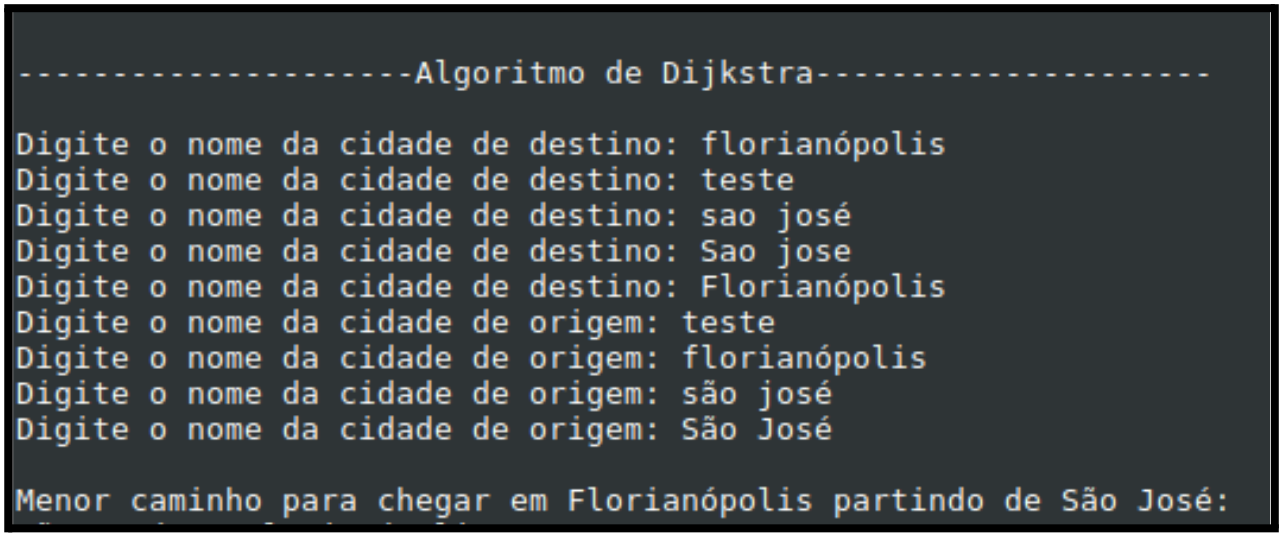
Não é necessário encaminhar um nome de arquivo como argumento da linha de comando, visto que o nome do arquivo que contém as rotas obrigatoriamente deve ser "**cities.csv**"

## 3. Execução do código para o usuário:

Inicialmente, o programa irá solicitar ao usuário o nome da cidade de destino, conforme ilustrado abaixo:




OBS: O nome da cidade de destino e de origem digitados precisam ser exatamente iguais a string correspondente no arquivo de rotas (.csv), caso o nome seja diferente do descrito neste arquivo, o programa irá solicitar ao cliente novamente a informação:




Após a inserção da cidade de origem e destino, o algoritmo irá calcular a melhor rota entre as cidades (a partir das distâncias entre cada par de cidades) e também a distância total entre as cidades solicitadas, e então retornar os valores no terminal, conforme a imagem abaixo:

```
----------------------Algoritmo de Dijkstra----------------------

Digite o nome da cidade de destino: Florianópolis
Digite o nome da cidade de origem: São José

Menor caminho para chegar em Florianópolis partindo de São José:
São José -> Florianópolis
Distância total: 19km
```

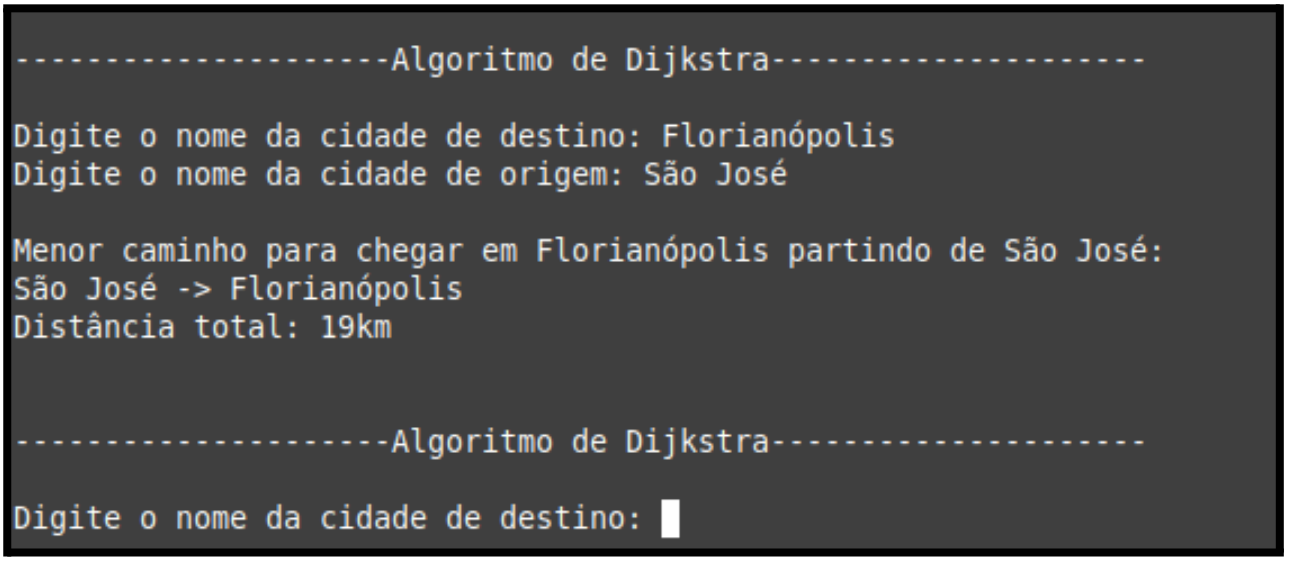
Em casos onde o usuário irá necessitar passar por mais de uma cidade, o algoritimo irá retornar a sequência de cidades partindo da origem para o destino, da esquerda para a direita:

```
----------------------Algoritmo de Dijkstra----------------------

Digite o nome da cidade de destino: Florianópolis
Digite o nome da cidade de origem: Santo Amaro da Imperatriz

Menor caminho para chegar em Florianópolis partindo de Santo Amaro da Imperatriz:
Santo Amaro da Imperatriz -> Palhoça -> São José -> Florianópolis
Distância total: 41km
```

Finalmente, com todas as informações já entregues ao usuário, o programa irá encerrar o ciclo do loop, então iniciar o próximo, onde irá solicitar novamente as informações para realizar outro cálculo, conforme a imagem abaixo:




## CONCLUSÃO

O programa conseguiu atingir todos os objetivos propostos para este trabalho, onde o usuário ao digitar no teclado as cidades de destino e origem recebe de volta a rota da sequência de cidades que terá que seguir para assim ter a menor rota possível.

O programa conseguiu atingir todos os objetivos propostos para este trabalho, onde foi criado a configuração de classes, a livre escolha de qual atendimento o cliente deseja e o pleno funcionamento da fila de atendimento seguindo os requisitos de tempo máximo de espera para cada classe e a ordem de prioridade dentre todas as classes.

## BIBLIOGRAFIA

- Sintaxes no acesso das informações contidas na tabela hash e também nas listas de estrutura:

Links revistos na implementação da tabela hash, em especial o link destacado em negrito:

**https://moodle.ifsc.edu.br/mod/book/view.php?id=311098&chapterid=52571**
https://moodle.ifsc.edu.br/mod/book/view.php?id=311098&chapterid=52572
https://moodle.ifsc.edu.br/mod/book/view.php?id=311098&chapterid=52573

- Revisão do algoritmo proposto para solução do problema **(visto diversas vezes ao longo do projeto)**:

Simulador: https://cmps-people.ok.ubc.ca/ylucet/DS/Dijkstra.html
Decritivo (site ufsc): http://www.inf.ufsc.br/grafos/temas/custo-minimo/dijkstra.html