



UTFPR - Universidade Tecnológica Federal do Paraná

Estrutura de Dados II

Simulador de Roteamento Utilizando Grafo

Alunos:

Gabriel Henrique Kwiatkowski Godinho 2126621

Erick José Teles de Andrade 2126575

Maio
2021

Conteúdo

1	Problema	1
2	Solução	2
2.1	Structs	2
2.2	Grafo Inicial	3
3	Funções	4
3.1	Conexões/Arestas	4
3.2	Cidades/Vértices	5
3.3	Busca de Rotas	5
3.4	Salvar e Carregar Grafo	9
4	Testes	10
5	Conclusão	13
	Bibliografia	14

1 Problema

Grafos são conjuntos finitos de vértices que podem ser conectados dois a dois por um conjunto de pares não ordenados, os quais são chamados arestas. Esta estrutura de dados possui diversas aplicações em problemas reais, entre elas, podemos destacar o uso da teoria dos grafos na representação de roteadores e enlaces através de um grafo $G=(N, A)$, onde os nós correspondem aos roteadores presentes na rede, e as arestas correspondem aos enlaces que conectam os roteadores.

Os algoritmos de roteamento existentes utilizam métricas para selecionar os caminhos percorrido entre dois pontos de uma rede, estas métricas podem ser o tamanho físico, a velocidade de transmissão da rede, a quantidade de arestas percorridas, o custo da rota devido aos pesos presentes nas arestas, dentre outros.

Devido a isto, a proposta do presente trabalho foi utilizar grafos ponderados para simular os possíveis caminhos de uma rede de roteamento de internet, visando resolver um problema de comunicação ente dois pontos de uma rede, para que os dados transportados por esta rede sejam levados de um ponto a outro pelo menor caminho existente. Sendo assim, para mostrar os custos das rotas foram utilizados os algoritmos de caminho mínimo de Dijkstra, e um algoritmo baseado na métrica de menor número de saltos.

2 Solução

Para resolver o problema e, assim, obter a melhor e mais rápida conexão entre dois pontos foi utilizada a estrutura de dados denominada grafo, e dentre os diferentes tipos de grafos foi escolhido o com lista de adjacências. Para se encaixar totalmente no empecilho e facilitar a visualização do usuário algumas modificações foram realizadas nas structs que compoem o grafo.

2.1 Structs

Na struct do vértice o inteiro *info* foi substituído por uma string *cidade* que armazena a sigla da cidade, o char *tipoC* por sua vez armazena o tipo de conexão estabelecida, (f) fibra óptica ou (c) cabo de ethernet.

```
typedef struct enlace {
    char cidade[SIGLA];
    char tipoC;
    struct enlace *next;
} Enlace;
```

Figura 1: Struct de um enlace

Já na struct do grafo, foi adicionada uma matriz de char *lista* que faz a relação de cada sigla de cidade com o seu índice, essa lista se faz necessária em funções que percorrem os vetores da lista de adjacência. Nela, o número da linha onde a sigla esta armazenada corresponde ao código da cidade, ou seja, a primeira cidade adicionada possui código 01, a segunda 02, etc...

```
typedef struct grafo {
    char **lista;
    int E;
    int V;
    Enlace **Adj;
} Grafo;
```

Figura 2: Struct de um grafo

2.2 Grafo Inicial

O simulador de roteamento é ideal para situações onde existem vários hosts e várias conexões, por este motivo, ao executar o programa, o usuário já possui em suas mãos cidades e conexões padrões, para conseguir realizar testes sem haver a necessidade da adição de vértice em vértice e aresta em aresta.

Esta configuração padrão segue o modelo da Rede Ipê, conexão em 2018:

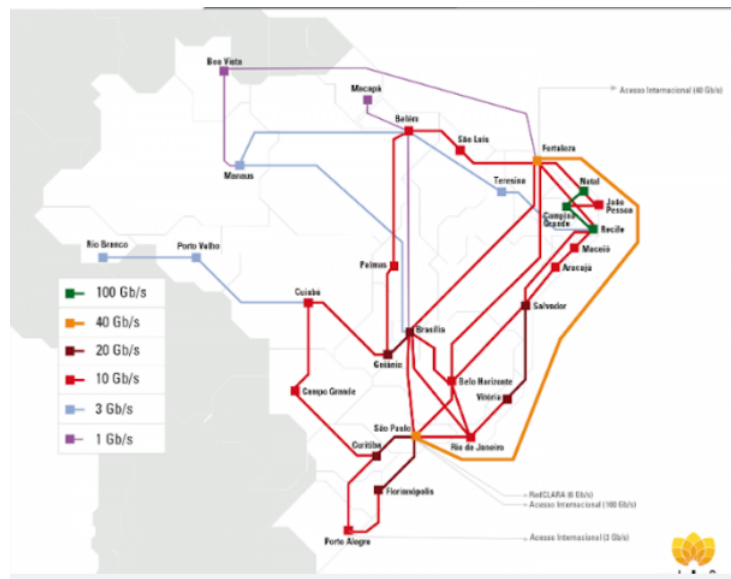


Figura 3: Conexão em 2018

Desse modo, para cumprir uma função mais de simulação e menos realista, as conexões de 10Gb/s, representadas no mapa pela cor vermelha, foram adicionadas como fibra óptica, enquanto as demais estão determinadas como cabo de ethernet, possuindo pesos 5 e 2, respectivamente.

3 Funções

É possível dividir o código do simulador em quatro blocos, cada um lidando com diferentes partes do problema e adicionando mais possibilidades de interações com o usuário, são eles: Conexões, Cidades, Busca de rotas e Armazenamento de dados.

Ao executar o programa, o usuário se depara com um menu, com várias opções partindo dele, estas opções estão divididas nos mesmos grupos para facilitar a utilização.

```
SIMULADOR DE NETWORK
MENU:
A. Adicionar conexao
B. Remover conexao
C. Alterar Conexao

D. Rota para todas as cidades
E. Melhor Rota entre duas cidades
F. Rota mais curta

G. Lista de Servidores

H. Adicionar Cidade
I. Remover Cidade
J. Limpar Simulacao

K. Salvar
L. Carregar
M. Sair
```

Figura 4: Menu do usuário

3.1 Conexões/Arestas

Neste primeiro bloco do menu são apresentadas funções que lidam com as conexões entre pontos do grafo, ou seja, as arestas. São ofertadas duas funções triviais, 'adicionar conexão' e 'remover conexão', porém foram alteradas para tratar com vértices contendo uma string de tamanho 3, a sigla da cidade que hospeda o ponto.

Além dessas, também é apresentada a função 'Alterar conexão', na qual o usuário consegue alterar o tipo de uma conexão já estabelecida, para cabo de ethernet (tipoC = 'c') ou fibra óptica (tipoC = 'f').

3.2 Cidades/Vértices

Nesta parte do menu o usuário possui 3 opções que envolvem as cidades do grafo, cidades as quais possuem um ponto de roteamento, estas são os vértices do grafo. Ao contrario das arestas, adicionar e remover vértices não são operações simples, pois conexões no meio de uma lista encadeada devem ser removidas e os vetores de cidades devem ser organizados e realocados a cada modificação.

Apesar das dificuldades, as 3 funções foram implementadas com sucesso, são elas: 'Adicionar cidade', 'Remover cidade' e 'Limpar grafo'. Esta última apaga todos os vértices e arestas do grafo, para um novo ser iniciado, pois quando o programa é executado algumas cidades e conexões entre elas são adicionadas automaticamente.

3.3 Busca de Rotas

Os algoritmos de roteamento possuem várias formas de classificação que refletem nas características de implementação e funcionamento de cada algoritmo.

Para o desenvolvimento deste trabalho foram utilizados algoritmos de roteamento global, pela necessidade de cada nó ter conhecimento sobre todos os outros nós da rede. Sendo assim, o algoritmo de roteamento global utilizado neste projeto foi o algoritmo de Dijkstra. Este método utiliza uma das formas mais tradicionais de roteamento, que consiste em encontrar o caminho mais curto entre dois nós de forma simples e eficaz. No algoritmo de Dijkstra, a partir de um nó de origem, é possível encontrar a menor rota até todos os outros nós conectados ao nó de origem. Outro método utilizado neste projeto, junto do algoritmo de Dijkstra, foi o protocolo de roteamento RIP, utilizando a métrica de menor número de saltos, pois este protocolo é usado quando não é possível atribuir peso aos links da rede, ou no caso do projeto, as arestas entre os nós.

```

while(cont < (G->V)-1){
    distMin = 99999;

    for(int i = 0; i < G->V; i++){
        if(dist[i] < distMin && !visitado[i]){
            distMin = dist[i];
            proxNo = i;
        }

        //verifica se existe melhor caminho atraves do proximo node
        visitado[proxNo] = 1;
        for(int i = 0; i < G->V; i++){
            if(!visitado[i])
                if(distMin + cust[proxNo][i] < dist[i]){
                    dist[i] = distMin + cust[proxNo][i];
                    pai[i] = proxNo;
                }
        }

        cont++;
    }
}

```

Figura 5: Função Dijkstra

No algoritmo de Dijkstra implementado neste trabalho, são enviados como parâmetros para a função o grafo criado dentro do projeto e os dois vértices dos quais se deseja encontrar o caminho mínimo entre eles. Dessa forma, a função inicia sua execução criando suas variáveis, uma das variáveis inicializada é a matriz de pesos do grafo, primeiramente esta matriz é iniciada com -1 em todas as posições da matriz, após isso é então feita a verificação das conexões do grafo, atribuído aos vértices com conexões o peso da sua conexão, que é definido a partir do tipo de conexão entre os pares ordenados, neste código foi utilizado o peso 2 para simular o custo de percorrer uma conexão em fibra óptica e 5 para percorrer uma conexão cabeada. Com a matriz de pesos inicializada, é possível fazer as atribuições restantes para as variáveis da função. Com as variáveis inicializadas corretamente, foi utilizado uma estrutura de repetição "while" para definir o menor caminho entre os nós.

Primeiramente, percorre-se o vetor de distâncias, que foi atribuído com os custos entre o vértice de entrada e o próximo vértice acessado, verificando se a distância da posição iterada é menor que a distância mínima encontrada até o próximo vértice, atualizando o valor da distância mínima nos casos de validação da condição. Após isso, é utilizado o vetor de posições visitadas para percorrer os caminhos ainda não avaliados entre os dois vértices, atualizando o vetor de distâncias sempre que a distância mínima somado ao custo de percorrimento até o próximo nó for menor que a distância atribuída ao vetor de distâncias.

Sempre que encontrado um valor que seja a verificado como a menor distância até o próximo nó, o índice do vértice é atribuída ao vetor pai, que ao final do código é usado para imprimir a rota utilizada para percorrer o caminho entre os dois vértices analisados. Uma possível saída correta da função é demonstrado na imagem de saída do terminal, nela podemos ver o caminho mínimo da conexão entre a cidade de Rio Branco e João Pessoa, as quais possuem os códigos de acesso 10 e 19 respectivamente.

```
-Digite '-1 X' para cancelar
-Digite o codigo de duas cidades para encontrar a melhor rota entre elas:
10
19
Custo da rota entre RBR e JOP = 23
Rota = RBR -> PVE -> CUB -> GOI -> BRA -> FOR -> NAT -> JOP
```

Figura 6: Saída da função Dijkstra

Outra função do projeto que utiliza exatamente a mesma lógica da função Dijkstra, no entanto, imprime o caminho, a partir de um vértice informado pelo usuário, até todos os vértices presentes na rede, demonstrando da mesma forma o caminho e o custo daquele percurso realizado. Esta função difere da função Dijkstra apenas no seu trecho de impressão, onde é usado um for para que seja possível imprimir os caminhos de conexão entre todos os vértices do grafo em relação ao vértice de entrada.

```
//imprime o caminho e o custo ate cada cidade
printf("\nCusto das rota a partir de %s:\n", G->lista[v]);
for(int i = 0; i < G->V; i++){
    if(i != v){
        printf("\nCusto da rota entre %s e %s = %d", G->lista[v], G->lista[i], dist[i]);
        printf("\nRota = %s", G->lista[i]);
        int aux = i;

        do{
            aux = pai[aux];
            printf(" <- %s", G->lista[aux]);
        }while(aux != v);
        printf("\n");
    }
}
```

Figura 7: Função Dijkstra para todos as conexões

Por fim, para criar um meio de comparação da eficácia dos métodos, foi criado uma função baseada no algoritmo de Dijkstra utilizando a métrica de menor número de saltos para percorrer um caminho entre dois vértices determinados pelo usuário. O desenvolvimento desta função se baseou em utilizar uma matriz de conexões entre os vértices sem o peso destas conexões, sendo assim, o programa encontra o caminho entre dois vértices buscando o menor número de passagens por vértices, independentemente do custo de aplicação deste método. Para exemplificar, as imagens trazem duas saídas utilizando os mesmos parâmetros de entrada, uma saída da função de caminho mínimo e outra saída da função baseada na utilização do método de menor número de saltos, em ambas as funções foi utilizado como os vértices de entrada a cidade de Porto Alegre, até a cidade de Natal, dessa forma as funções retornaram saídas diferentes para cada método, o que condiz com o esperado de cada método.

```
-Digite '-1 X' para cancelar
-Digite o código de duas cidades para encontrar a melhor rota entre elas:
0
20

Custo da rota entre POA e NAT = 13
Rota = POA -> FLO -> SPO -> BHO -> FOR -> NAT
```

Figura 8: Saída da função de caminho mínimo

```
-Digite '-1 X' para cancelar
-Digite o código de duas cidades para encontrar uma rota entre elas:
0
20

Rota entre independente do custo NAT e POA
Rota = POA -> FLO -> SPO -> FOR -> NAT
```

Figura 9: Saída da função de menor número de saltos

3.4 Salvar e Carregar Grafo

Um problema encontrado durante o desenvolvimento do programa foi o armazenamento de dados, muitas vezes adicionavam-se cidades que não estavam na lista inicial para realização de testes, porém, ao fechar o simulador e abri-lo novamente as cidades eram apagadas. Então foram implementadas as funções 'Salvar grafo' e 'Carregar grafo', a primeira armazena todas as cidades atuais e suas conexões em um arquivo de texto '*save.txt*', enquanto a segunda por sua vez apaga o grafo que está sendo utilizado no programa e o substitui pelo armazenado no arquivo de texto.

4 Testes

Para criar uma seção de testes de fácil visualização, foi criado um grafo menor com 3 vértices, e a partir deste grafo, foram executadas as funções disponíveis no código.

Inicialmente listamos o grafo criado para a sessão de teste:

```
[00] CWB-> SPO (Tipo: f)
[01] SPO-> CWB (Tipo: f)  FOR (Tipo: f)
[02] FOR-> SPO (Tipo: f)
```

Figura 10: Cidades iniciais e suas conexões

Primeiramente, a cidade de Florianópolis(FLO) foi adicionada, assim como uma conexão entre a própria e Curitiba(CWB) por meio de cabo ethernet:

```
-Digite 'z' para cancelar
-Digite a sigla da cidade a ser inserida (3 letras, todas maiusculas): FLO
```

Figura 11: Cidades iniciais e suas conexões

```
-Digite 'X X z' para cancelar
-Digite o codigo das duas cidades a serem conectadas e o tipo de conexao
('f' para fibra optica e 'c' para cabo ethernet):
3 0 c
```

Figura 12: Conexão entre FLO e CWB sendo adicionada

```
[00] CWB-> FLO (Tipo: c)  SPO (Tipo: f)
[01] SPO-> CWB (Tipo: f)  FOR (Tipo: f)
[02] FOR-> SPO (Tipo: f)
[03] FLO-> CWB (Tipo: c)
```

Figura 13: Cidades e conexões após inserções

Em seguida, utilizando a opção 'E. Melhor rota entre duas cidades', foi encontrado caminho com menor custo entre a cidade adicionada Florianópolis(FLO) e a cidade de Fortaleza(FOR):

```
Custo da rota entre FOR e FLO = 9
Rota = FLO -> CWB -> SPO -> FOR
```

Figura 14: Melhor rota entre FLO e FOR

Logo após, a configuração atual foi salva em um arquivo de texto por meio da opção 'K. Salvar' e a conexão entre FLO e CWB foi removida:

```
-Digite '-1 X' para cancelar
-Digite o código das duas cidades a serem desconectadas:
0 3
```

Figura 15: Removendo a conexão FLO - CWB

```
[00] CWB-> SPO (Tipo: f)
[01] SPO-> CWB (Tipo: f)  FOR (Tipo: f)
[02] FOR-> SPO (Tipo: f)
[03] FLO->
```

Figura 16: Lista de cidades e conexões neste passo

Assim como a conexão, a cidade de Florianópolis também foi removida:

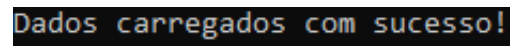
```
-Digite '-1' para cancelar
-Digite o código da cidade a ser removida:
3
```

Figura 17: Remoção da conexão FLO - CWB

```
[00] CWB-> SPO (Tipo: f)
[01] SPO-> CWB (Tipo: f)  FOR (Tipo: f)
[02] FOR-> SPO (Tipo: f)
```

Figura 18: Lista de cidades e conexões neste passo

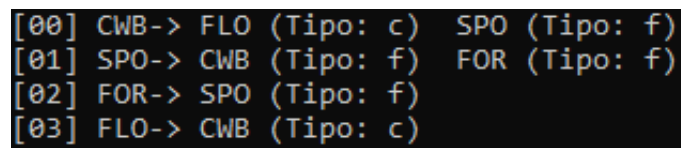
Finalmente, para encerrar os testes, foi utilizada a opção 'L. Carregar' para recuperar a cidade e a conexão excluídas:



```
Dados carregados com sucesso!
```

A terminal window with a black background and green text displaying the message "Dados carregados com sucesso!".

Figura 19: Dados salvos foram carregados



```
[00] CWB-> FLO (Tipo: c) SPO (Tipo: f)
[01] SPO-> CWB (Tipo: f) FOR (Tipo: f)
[02] FOR-> SPO (Tipo: f)
[03] FLO-> CWB (Tipo: c)
```

A terminal window with a black background and green text displaying a list of four connections between cities. Each line shows an index in brackets, followed by a city name, an arrow, another city name, and then two parenthetical entries with labels and types.

Figura 20: Lista de cidades e conexões final

5 Conclusão

O desenvolvimento do trabalho atendeu os requisitos estabelecidos e teve todas as suas funções desenvolvidas plenamente. A utilização de grafos para simular o roteamento de uma rede se mostrou eficaz e o uso de algoritmos de caminho mínimo obteve resultados satisfatórios ao demonstrar as possíveis rotas de uma conexão. A criação de um grafo baseado em um mapa existente ajudou durante os testes do código, comprovando a eficácia das funções. O uso de uma estrutura complexa como o grafo, e a implementação de funções complexas agregaram para aumentar o conhecimento dos alunos sobre o conceito de grafos e principalmente de roteamento utilizando caminhos mínimos. Dessa forma, o desenvolvimento do projeto foi completo e eficaz.

Bibliografia

CROSSETI, Melissa Cruz. O que é um Backbone?. **Tecnoblog**, 2018. Disponível em: <https://tecnoblog.net/277282/o-que-e-um-backbone/> Acesso em: 07 maio 2021

SOARES, Samuel Alves. **Simulação de algoritmos de roteamento Unicast utilizando o simulador NS-2**. Faculdade Farias Brito, Fortaleza, 2009.