Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Estrutura de Dados – 1/2016
Alunos: André Fernandes Freire 15/0005539
Pedro Vitor Fonseca 15/0064012

Documentação

1. Desenvolvimento

O programa recebe os dados de um arquivo txt e os armazena em um grafo. Cada vértice do grafo corresponde à uma cidade e cada aresta corresponde aos caminhos entre as cidades, os quais variam de tamanho.

O objetivo é buscar o menor caminho que passe por todas as cidades apenas uma vez, para isso foi usado uma alteração do algoritmo de busca por profundidade. O programa fará uma busca por todo o grafo, o marcando com cores brancas ou cinzas, as cidades marcadas com a cor branca são cidades que ainda não foram visitadas e as cidades marcadas com cinza foram visitadas uma vez já.

Sempre que o algoritmo chega em uma cidade em que todas as cidades vizinhas já foram visitas (cinzas), ele verifica se o caminho percorrido passou por todas as cidades, caso tenha passado ele compara com um valor anteriormente marcado para o menor caminho e , se o caminho encontrado for menor, o altera. Além disso ele modifica um vetor que contém a ordem das cidades visitadas.

Após fazer essa verificação, o algoritmo marca a cidade atual com a cor branca e volta para a cidade anterior para continuar a verificação em todos os caminhos possíveis.

Um decisão importante no algoritmo foi podar algumas verificações desnecessárias, pois verificar todos os caminhos possíveis é bem caro. Para fazer essa “poda” foi incluída uma verificação no início da recursão, que analisa se o caminho percorrido até então já não é maior que o menor caminho encontrado até então, fazendo com que o algoritmo retorne antes mesmo de analisar todo o caminho, poupando assim algumas verificações.

Para os testes de desempenho foi usada a função gettimeofday, que verifica o tempo de execução do programa. Além disso foram usadas diferentes entradas pro programa, com o objetivo de analisar a dependência entre o tempo de execução e o número de vértices e arestas do grafo.

2. Descrição dos módulos

No programa foram criados apenas três módulos, cada um contendo seu módulo de implementação e de definição.

O primeiro módulo (grafo) contém a estrutura de dados utilizada no programa, para isso foram definidas as seguintes estruturas:

- TipoApontador: Um ponteiro para um outro nó das listas de adjacência.
- TipoNoAdj: Contém um nó da lista de adjacências, esse nó guarda a cidade vizinha e a distância entre ela e a cidade de origem.
- TipoListaAdj: Contém o ponteiro inicial da lista de adjacências e a cor que informa se o caminho para a cidade X já foi percorrido ou não.
- TipoVertice: Estrutura que engloba todas as outras e que contém os vértices do grafos e todas as suas informações.

Além dessas estruturas, o módulo contém as funções abaixo, que são próprias da estrutura de dados “grafo” e permitem algumas operações no grafo.

1. FGVazia: Inicia o grafo, alocando memória para a estrutura tipo vértice além de denotar uma cor para e fazê-lo apontar para NULL.
2. InserirAresta: Insere um “caminho” entre os vértices, adicionando um nó na lista de adjacências.
3. PreencherGrafo: Função que recebe os dados do grafo de um arquivo texto e os insere no grafo.
4. EsvaziarGrafo: Desaloca a memória usada pelo grafo, deixando-o vazio.

O segundo módulo (busca) é responsável pela busca do melhor caminho passando por todos os vértices do grafo e pela impressão da saída do programa. Nele estão contidos as seguintes funções:

1. VerificarCidades: Verifica se o caminho encontrado passou por todas as cidades, retorna 1 caso seja verdadeiro e 0 caso seja falso.
2. CopiarVetor: Copia os dados que estão contido em um vetor para outro.
3. DFS: Função que faz a pesquisa em profundidade no grafo, a fim de encontrar o menor percusso dentre os possíveis.
4. ImprimeSaida: Imprime a saída do programa na tela, mostrando as distâncias parciais, o caminho a ser percorrido e a distância total.
5. BuscarMT: Função que inicializa a busca no grafo e chama a função que imprime a saída.

O terceiro módulo (main) recebe as entradas, chama as funções que inicializam o grafo e que executam a busca, além de chamar uma função para esvaziar o grafo. Ele contém as seguintes funções:

1. show_help: Mostra uma tela de ajuda ao usuário.
2. main: Executa todo o programa.

3. Descrição dos TAD's

No programa foi utilizado o TAD grafo e sua implementação foi dada através de um vetor de listas de adjacências. Cada registro do vetor corresponde a um vértice e contém um ponteiro para o primeiro elemento da lista de adjacências e a cor do vértice. A lista de adjacências vai conter os vértices vizinhos e a distância entre eles, compondo assim as arestas do grafo.

As operações executadas no grafo foram: a inicialização do grafo, a inserção de arestas, o preenchimento dele com os dados contidos em um arquivo de texto, a busca do menor caminho e o esvaziamento do grafo.

4. Entrada de dados

A entrada do programa consiste em um arquivo de texto contendo o número de cidades e os caminhos entre as cidades. O número de cidades era informado na primeira linha do arquivo e as linhas abaixo continham a cidade de origem, a cidade de destino e a distância entre elas, esses três dados são separados por espaço e ocupam uma linha.

Para o programa fazer a entrada dos dados do arquivo de texto o nome dele deve ser informado em linha de comando, além disso deve-se informar também a cidade de origem de um dos amigos.

5. Saída de dados

A saída será impressa na tela assim que o programa terminar de ser executado, apresentando ao usuário as distâncias parciais entre cidades, o melhor caminho e a distância total a ser percorrida. Na tela as informações aparecerão como no modelo abaixo:

Distância entra as cidades A e B: X

…

Distância entra as cidades Y e Z: W

Caminho mais curto: A-B-...-Y-Z

Distância total: N

6. Explicação sobre como utilizar o programa

Para fazer a compilação digite na linha de comando “make”, depois para executar o programa digite ./grafo -e <nomedoarquivo> -o <origem> , onde a opção -e faz a entrada do nome do arquivo de texto contendo as entradas do programa, para isso substitua <nomedoarquivo> pelo nome do arquivo. A opção -o informa o local de onde o grafo vai começar a ser analisado, para isso substitua <origem> pela cidade de origem.

7. Complexidade do algoritmo

1. Grafo:

    1.1. FGVazia: O(numCidades), pelo fato da função colocar o apontador e a cor branca para todos os vértices(quantidade de cidades).
    1.2. InserirAresta: O(1), pela ausência de loop, logo todos as linhas são executados apenas uma vez por chamada da função.
    1.3. PreencherGrafos: O(n), n é igual a quantidade de linhas presentes no arquivo .txt, já que nessa função recebe as informações para criar o grafo, e a função presente dentro dela é O(1).

2. Busca:
    2.1. VerificarCidades: O(numCidades), no pior caso ele ira percorrer todos os vértices e verificas se todos estão com a cor adequada.
    2.2. CopiaVetor: O(numCidades), faz a cópia de um vetor de tamanho correspondente ao número de cidades.
    2.3. DFS: O(numCidades²), há um loop que executa a função por toda a lista de adjacências e cada execução da função deve verificar todos os vétices. Logo temos que o número de chamadas da função é o número de vértices e a cada chamada ele verifica todos os vértices e então a complexidade é O(numCidades).
    2.4. ImprimeSaida: O(numCidades), imprime as escolhas feitas pelo programa de melhor percuso logo, ele mostrara os percursos entre a cidades, sendo numCidade-1, e como constantes são desprezadas, numCidades.
    2.5. BuscaMT: O(numCidade²), pelo fato de chamar DFS e ImprimeSaida.

3. Main: O(numCidades²), pois chama a função BuscaMT e outras funções menos complexas.

8. Análise de resultados

Para a análise de resultado foram consideradas 5 entradas diferentes, variando o número de cidades, mas ainda assim considerando grafos esparsos.

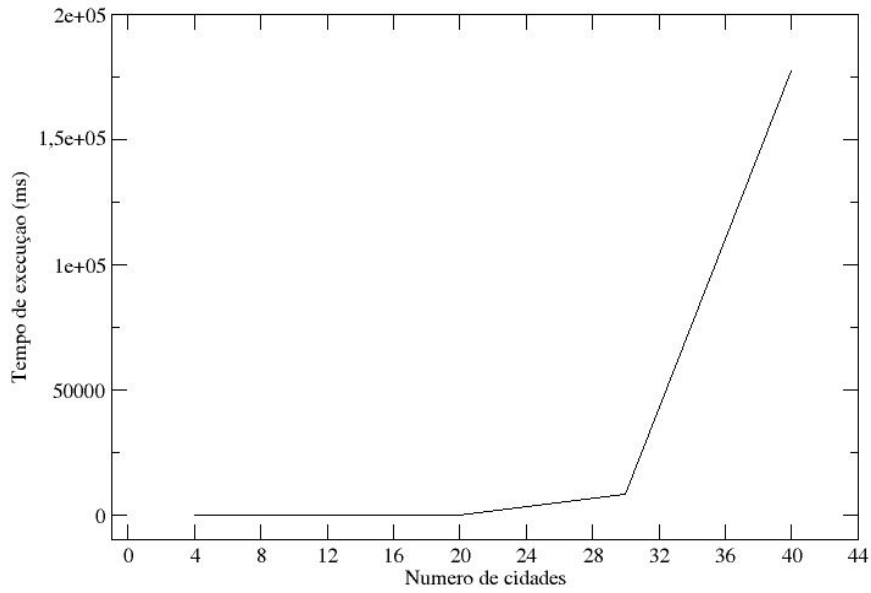
O programa foi executado para cada entrada e através da função gettimeofday o tempo de execução foi medido, variando a cidade de origem e fazendo uma média entre os tempos de execução encontrados para cada entrada. Com isso pode-se obter a tabela abaixo com o tamanho das entradas e os respectivos tempos de execução:

Tabela 1 - Desempenho algoritmo com poda

Número de cidades	Tempo de execução (ms)
4	0
10	0
20	113
30	8163
40	177358

Pela tabela pode-se perceber o crescimento do tempo de execução em função do número de cidades. Mas para observar melhor, temos o gráfico abaixo:

Grafico 1 - Tempo de execuçao vs nº de cidades



A partir do gráfico acima pode-se perceber que há uma relação quadrática ou talvez exponencial entre o tempo de execução e o número de cidades.

Além da análise acima, outra análise pode ser feita comparando o desempenho do algoritmo quando usando algum método que reduza o número de recursões, ou seja, uma poda e quando o código não apresenta esse método.

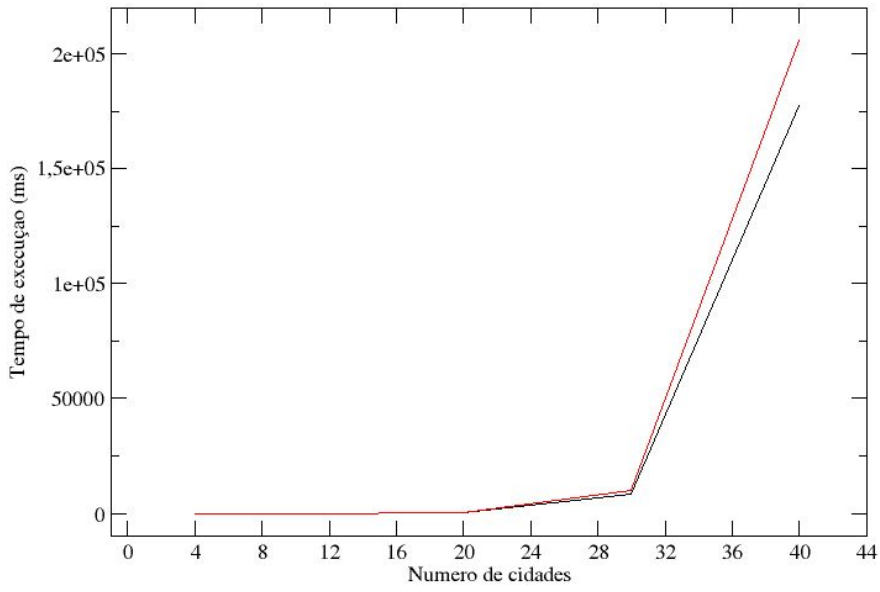
A análise acima foi feita com o código instrumentado para não entrar em uma recursão caso a distância de um caminho se mostre maior do que a menor já armazenada. Executando o código sem essa poda na execução, obteve-se os seguintes dados:

Tabela 2 - Desempenho algoritmo sem poda

Número de cidades	Tempo de execução (ms)
4	0
10	0
20	142
30	10061
40	205674

O gráfico abaixo mostra as curvas dos tempos de execução versus o número de cidades do algoritmo sem poda e do algoritmo com poda:

Grafico 2 - Tempo de execuçao vs nº de cidades



Como pode-se notar no gráfico, o uso de técnicas de podagem no algoritmo pode reduzir o tempo de execução de forma considerável, ainda mais se o grafo for muito denso.

Apesar de existirem métodos para reduzir o tempo de execução desse tipo de problema, em todos os casos serão feitas análises de grande parte das arestas, fazendo com que o programa percorra quase todos os caminhos, ou em um pior caso, todos os caminhos possíveis.