Universidade Federal de Minas Gerais

ICEx/DCC

Algoritmos e Estruturas de Dados III

Trabalho Prático 2

Aluno: Alexandre Torres Pimenta

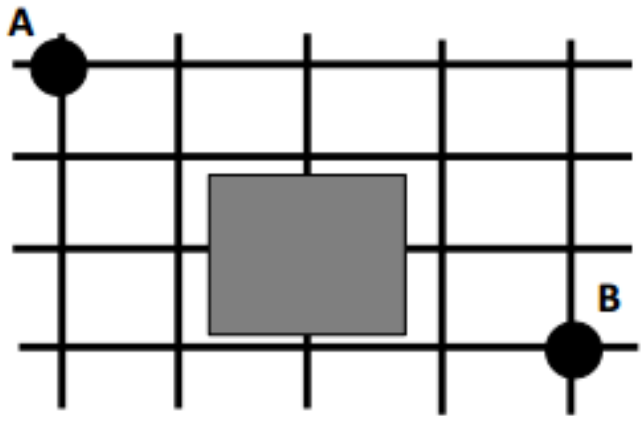
Belo Horizonte, 24 de Outubro de 2014

Introdução

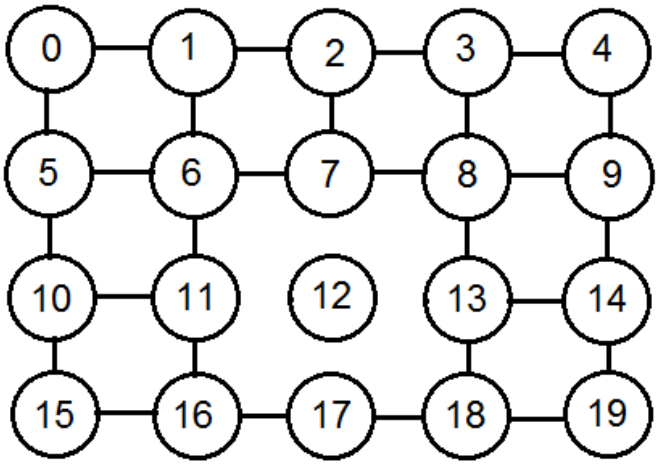
O objetivo deste trabalho prático é implementar um algoritmo para calcular o número de menores caminhos possíveis em uma cidade em que as ruas e avenidas são dispostas no formato de uma grade, em que as esquinas podem ou não ser bloqueadas. Na vida real, um algoritmo do tipo pode ter várias aplicações práticas, em áreas como roteamentos de caminhos, e solução de labirintos.

Implementação

Para desenvolver o algoritmo, a cidade foi modelada como um grafo não direcionado, e sem pesos, como podemos ver na figura:

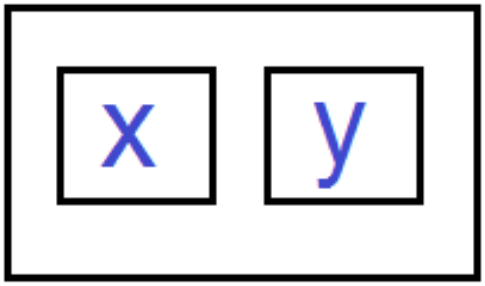


(Cidade, como representada na documentação)

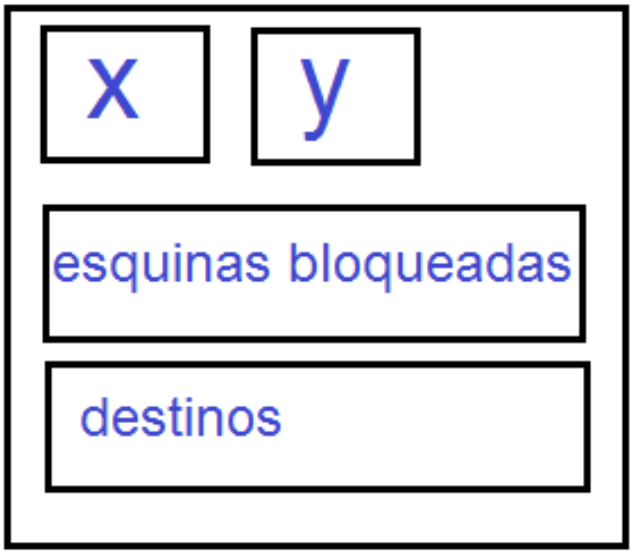


(Grafo direcionado e sem pesos, representando a mesma cidade)

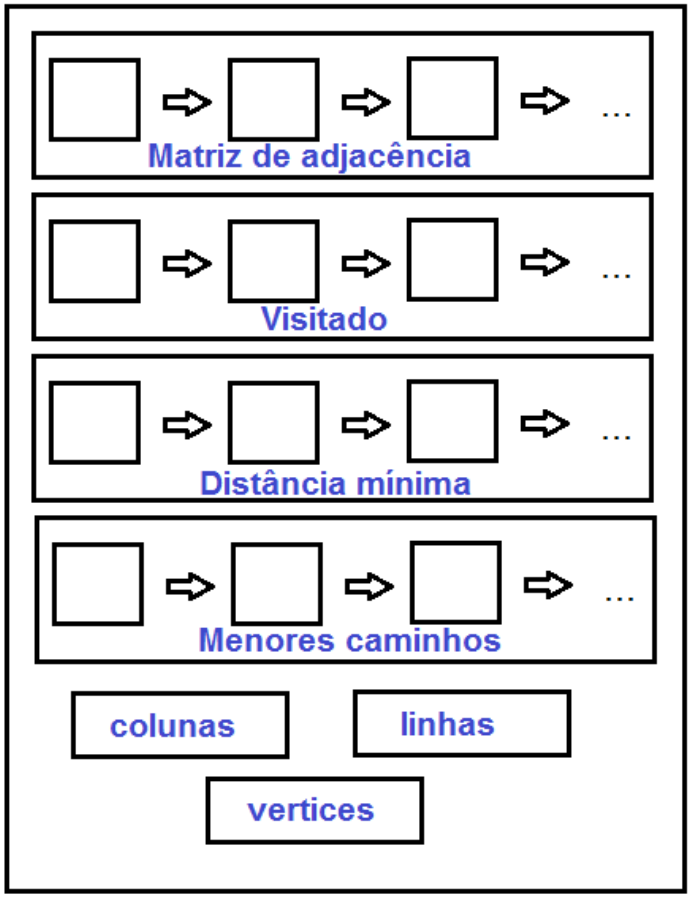
Para determinar o número de menores caminhos possíveis do vértice 0, que sempre vai ser o vértice de origem, para um outro vértice qualquer, foi aplicada uma busca em largura, que à medida que percorre o grafo, armazena em cada vértice o número de menores caminhos até ele, e a menor distância até ele. O tipo abstrato grafo foi implementado usando matriz de adjacência. Para a busca em largura, foi utilizado o tipo abstrato de dado fila, e para a cidade, foram criados dois tipos abstratos de dado, o TAD Cidade e o TAD Esquina.



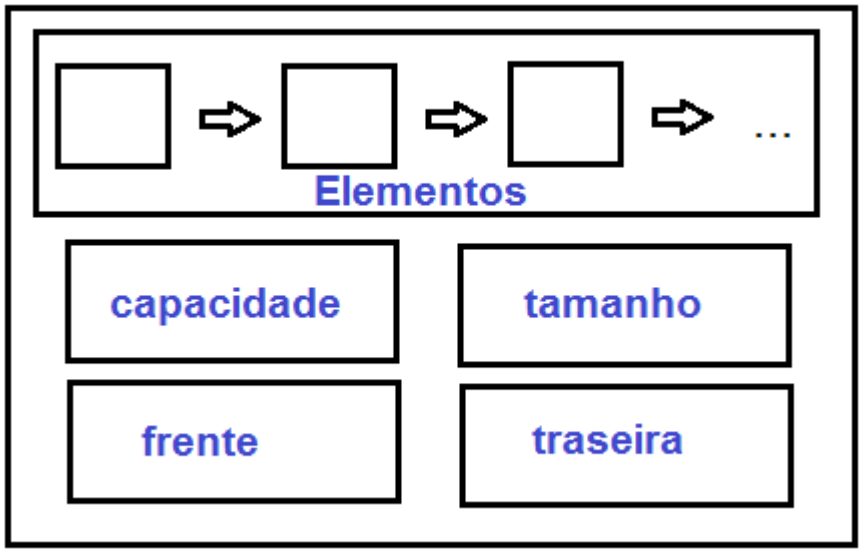
Representação gráfica do TAD Esquina. O TAD contém dois inteiros, X e Y, para indicar em qual rua e avenida da cidade a esquina representada está.



Representação gráfica do TAD Cidade. O TAD contém 4 inteiros. X e Y para indicar as dimensões da cidade (Ruas x Avenidas), um inteiro para indicar a quantidade de esquinas bloqueadas, e mais um para indicar o números de destinos à serem calculados por instância de execução.



Representação gráfica do TAD Grafo. Esta é a estrutura de dados mais importante do programa. Ela contém 4 ponteiros para vetores de inteiros. Um de tamanho Vértices x Vértices que é usado para simular a matriz de adjacência do grafo, um de tamanho Vértices para indicar se cada vértice já foi acessado ou não, um de tamanho Vértices que indica a distância mínima desde a origem para cada vértice, e um de tamanho Vértices para indicar o número de menores caminhos para cada vértice desde a origem. O TAD tambem contém 3 inteiros, um para representar a quantidade de vértices no grafo, e outros dois para indicar a quantidade de linhas e colunas.



Representação gráfica do TAD Fila. Este é o TAD utilizado na busca em largura. Ele contém um ponteiro para o vetor de inteiros que possui os elementos da fila, e 4 inteiros, para representar a capacidade, tamanho, a frente e traseira da fila.

Funções e procedimentos

int matrixOffset(int Columns, int i, int j)

É a função usada para simular uma matriz dentro de um vetor. Ela recebe a posição i,j, e retorna um inteiro que indica onde a posição i,j da matriz está no vetor.

void insertEdge(int source, int dest, Graph *G)

A função insere uma aresta na matriz de adjacência, do vértice source até o vértice dest.

void initializeGraph(Graph *G, City C)

Função usada para inicializar o grafo. Ela insere o número de colunas e linhas no grafo, aloca o espaço para a matriz de adjacência, e inicializa a matriz com 0 em todas as posições.

int translateToVertex(int X, int Y, City C)

A função traduz o par de coordenadas X,Y que representa uma esquina da cidade, para um inteiro que representa qual vértice representa esta esquina no grafo.

void removeVertex(int vertex, Graph *G)

A função remove todos as arestas do vértice representado pelo inteiro vertex na matriz de adjacência do grafo.

int borderCheck(int Vertex, Graph G)

A função verifica, usando as propriedades do grafo, se o vértice recebido é uma das 4 extremidades do grafo, ou se ele está na borda do grafo.

void buildGrid(Graph *G)

A função constrói o formato de grade da cidade no grafo, usando a função ***borderCheck***. Ela determina onde o vértice está a cada iteração, e o insere na matriz de adjacência usando a função ***insertEdge***.

Queue *CreateQueue(int maxElements)

A função aloca o espaço para os elementos da fila, inicializa suas propriedades, e retorna o ponteiro indicando a primeira posição da fila criada.

void Dequeue(Queue *Q)

A função remove um elemento da fila.

int Front(Queue *Q)

A função retorna o elemento que está na frente da fila.

void Enqueue(Queue *Q,int element)

A função insere um elemento na fila.

void shortestPaths(Graph *G)

A função aloca e inicializa os vetores de visitados, mínima distância e caminhos mais curtos, e inicia a busca em largura.

void BFS(Graph *G, int Source)

O funcionamento da função é explicado pelo seguinte pseudo-código:

```
BuscaEmProfundidade(Grafo, Source)
{
    Criar fila
    Inserir o vértice Source na fila

    Enquanto a fila não estiver vazia, faça:
    {
        Source é igual ao primeiro elemento da fila
        Desenfileirar

        Para cada vértice vizinho ao vértice Source, faça:
        {
            Se o vértice ainda não for visitado
            {
                Marcar o vértice como visitado
                Distância mínima até o vértice é igual distância mínima desde Source mais um
                Número de menores caminhos é igual ao número de menores caminhos desde Source
                Enfileirar vértice
            }
            Se o vértice já for visitado
            {
                Se a distância mínima do vértice for igual à distância mínima de Source
                {
                    Somar número de menores caminhos até Source ao número de menores caminhos até o vértice
                }
                Se a distância mínima do vértice for maior do que a distância mínima de Source
                {
                    A distância mínima do vértice é igual à distância mínima de Source
                    O caminho de menores caminhos até o vértice é igual ao número de menores caminhos até Source
                }
            }
        }
}
```

Organização do código e detalhes técnicos

O código está separado em três arquivos. Os arquivos grafo.h e grafo.c são usados para implementar os tipos abstratos de dados e suas operações, e o arquivo main.c implementa o programa principal. O compilador utilizado foi o GNU CCC COMPILER no sistema operacional Microsoft Windows 8.1.

Análise de complexidade

int matrixOffset(int Columns, int i, int j)

A função realiza sempre apenas uma operação, então sua complexidade é constante: $O(1)$

void insertEdge(int source, int dest, Graph *G)

A função acessa diretamente a matriz de adjacência e insere a aresta. Então sua complexidade é constante: $O(1)$

void initializeGraph(Graph *G, City C)

A função inicializa os valores do grafo, aloca memória para a matriz de adjacência, e preenche toda a matriz com 0. Por percorrer toda a matriz, a complexidade da função é $O(v^2)$, onde v é o numero de vértices. v também é equivalente ao número de ruas vezes o número de avenidas.

int translateToVertex(int X, int Y, City C)

A função realiza sempre apenas uma operação, então sua complexidade é constante: $O(1)$

void removeVertex(int vertex, Graph *G)

A função vai percorrer uma coluna, e uma linha da matriz de adjacência, removendo as arestas do vértice. Então, sua complexidade será $O(2v)$, onde v é o número de vértices.

int borderCheck(int Vertex, Graph G)

No pior caso, a função irá fazer as 8 comparações, e retornar o caso 8, então a complexidade da função no pior caso é $O(8)$.

void buildGrid(Graph *G)

A função irá iterar todos os vértices, adicionando suas arestas na matriz de adjacência, então sua complexidade será $O(v)$, onde v é o número de vértices.

Queue *CreateQueue(int maxElements)

A função terá complexidade constante,pois sempre irá realizar as mesmas operações (inicializar a fila). Por realizar sempre 6 atribuições, sua complexidade é $O(6)$.

void Dequeue(Queue *Q)

No pior caso, a função irá realizar 3 atribuições. Então sua complexidade é constante: $O(3)$.

int Front(Queue *Q)

A complexidade da função é constante, pois ela sempre retorna o elemento na primeira posição. $O(1)$.

void Enqueue(Queue *Q,int element)

No pior caso, a função irá realizar 4 atribuições. Então sua complexidade é constante, $O(4)$.

void shortestPaths(Graph *G)

A função irá alocar memória para os 3 vetores de operações da função BFS, e percorrê-los, e então chamar a função de busca em largura. Então sua complexidade será $O(4v + a)$, onde v representa o número de vértices e a o número de arestas.

void BFS(Graph *G, int Source)

No pior caso, a função irá acessar todos os vértices, e passar por todas as arestas do grafo. Então sua complexidade é $O(v + a)$, onde v representa o número de vértices, e a o número de arestas. O número de arestas pode variar de acordo com quantas esquinas estão bloqueadas na cidade em cada instância.

Complexidade do programa principal

Desconsiderando operações e atribuições constantes que são assintóticamente insignificantes, o programa principal faz a leitura do arquivo de entrada, então para cada instância, aloca e inicializa a matriz de adjacência do grafo (v^2), depois popula a matriz de adjacência com arestas para formar o padrão da cidade (v), então para cada esquina bloqueada, ele passa por uma linha e uma coluna da matriz de adjacência removendo as arestas (2v), e então faz a busca em largura

(4v+a). Considerando tudo isso, para cada instância o programa principal tem a complexidade $O(v^2 + 7v + a)$. Então a complexidade final, é de $O(i(v^2 + 7v + a))$, onde i representa o número de instâncias, v o número de vértices, e a o número de arestas. O número de vértices é equivalente ao número de ruas vezes o número de avenidas, e o número de arestas varia de acordo com quantas esquinas estão bloqueadas.

Análise espacial

Desconsiderando inteiros usados para iteração, e a string usada para armazenar dados lidos do arquivo de entrada que são assintóticamente insignificantes, podemos categorizar o consumo de memória do programa nos seguintes itens:

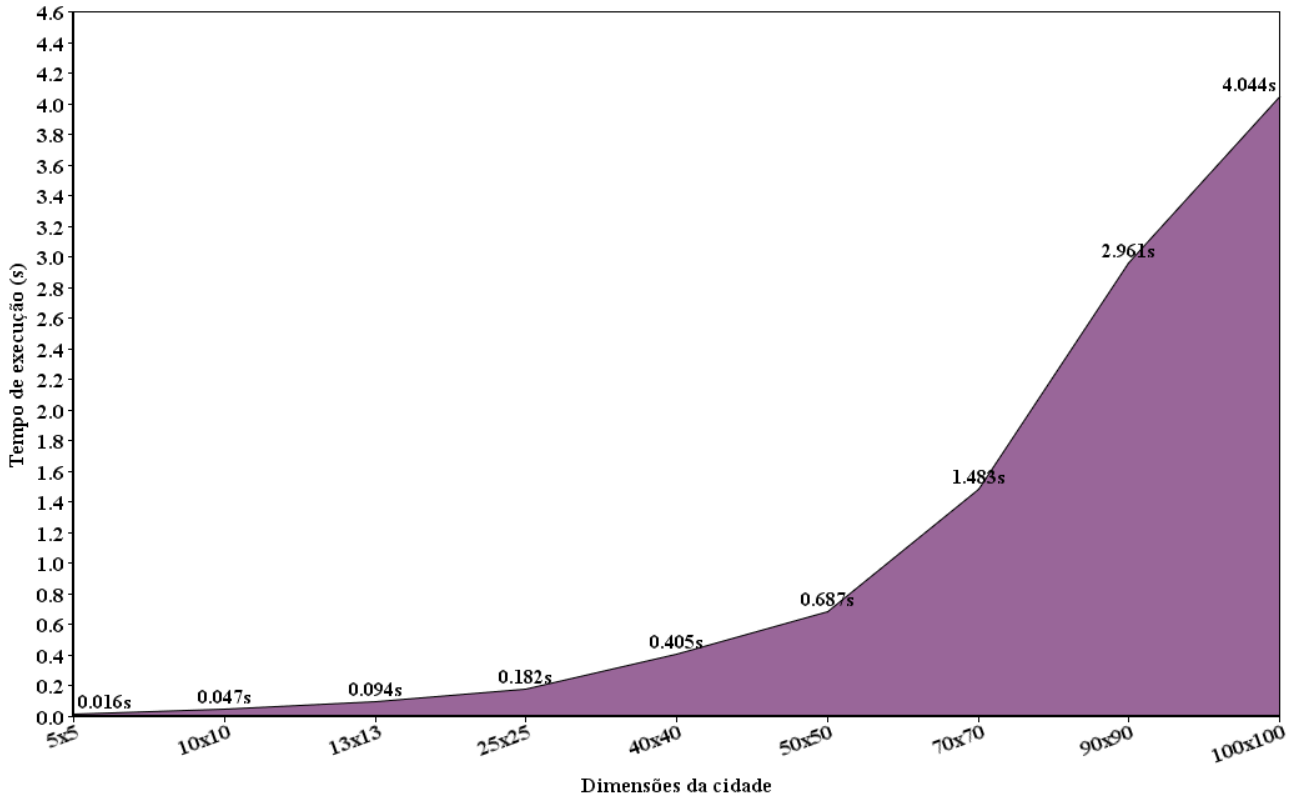
- A matriz de adjacência do grafo, que têm tamanho vértices x vértices
- Os três vetores de operação do TAD grafo, cada um com tamanho vértices
- A fila usada na busca em largura, também com tamanho vértices

Levando isso em conta, podemos determinar que o gasto espacial do programa é regido por $\Omega(v^2 + 4v)$.

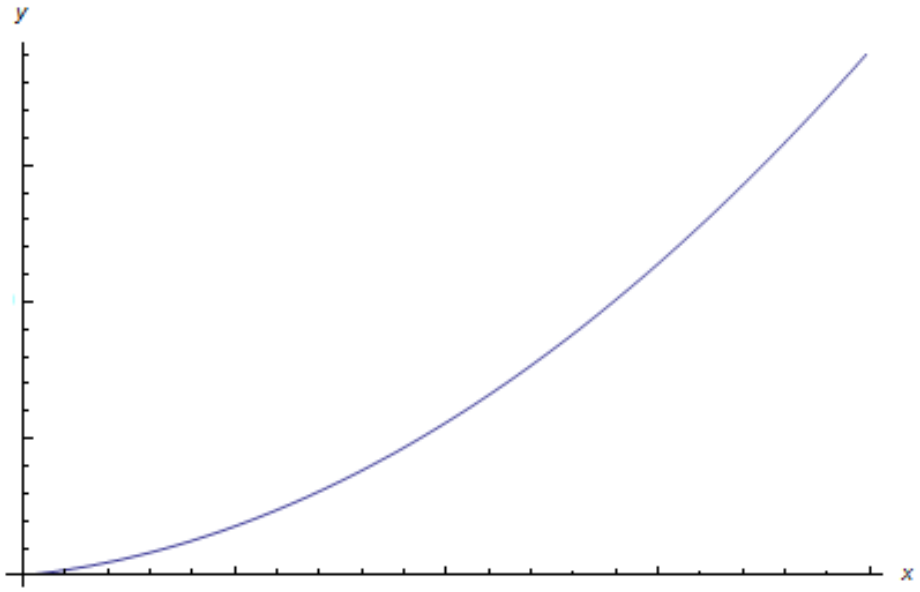
Para casos em que as dimensões da cidade são pequenas, como os casos propostos na documentação e arquivos de testes, a solução apresentada é eficiente, como pode se notar nos testes. Porém, para casos em que as dimensões da cidade são muito grandes, o consumo de memória e tempo de execução explodem exponencialmente. Por exemplo, em uma cidade com dimensão 25x25, a tarefa é executada em 0.182s, com o consumo de memória de aproximadamente 800 kilobytes (considerando cada inteiro como 2 bytes). Por outro lado, uma cidade com dimensão 100x100, leva 4.044s para executar, e consome 200 megabytes. A partir disso, os números explodem. Uma cidade com dimensão 200x200 gastaria aproximadamente 3.2 gigabytes, e se levarmos as dimensões a um extremo, como 90000x90000, teremos um gasto gigantesco de 131 exabytes. Para reduzir drasticamente esse gasto de memória, pode se usar listas de adjacência no lugar da matriz de adjacência. Porém para os casos propostos (em que a cidade tem uma dimensão pequena), a matriz de adjacência é suficiente, e mais eficiente.

Testes

Os testes foram realizados usando um AMD Phenom II X4 840, com 8 GB de memória RAM, no sistema operacional Microsoft Windows 8.1.



No gráfico, podemos ver que o comportamento assintótico do programa é quadrático, e seu gráfico segue o padrão do gráfico da sua função de complexidade:



Conclusão

A solução proposta conseguiu realizar com sucesso todas as operações propostas na documentação e arquivos de teste, em tempo ágil e com baixo consumo de memória. Como vimos nas análises, se a dimensão da cidade for muito grande, essa solução não seria ideal, pois ela explode exponencialmente. Para estes casos, o ideal é usar listas de adjacência no lugar da matriz de adjacência.

Referências

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição.

http://stackoverflow.com/questions/15211611/number-of-shortest-paths-in-a-graph

http://en.wikipedia.org/wiki/Breadth-first_search

Anexos

Listagem dos códigos:

- main.c
- grafo.c
- grafo.h