Trabalho Prático 1 - Vizinhança Errada

Julio Cesar Tadeu Guimarães - 2012049669

22 de Setembro de 2013

1. Introdução

Este trabalho traz um problema comum encontrado em várias situações, um problema de caminhamento de grafos. Neste trabalho temos uma vizinhança composta por alguns quarteirões e outras ruas que ligam estes quarteirões. Porém essa cidade está com um alto índice de criminalidade, assim tendo como base a probabilidade de ser assaltado em cada rua que se passa, o entregador está querendo saber qual é o melhor caminho para chegar em algum quarteirão partindo de outro. Para diminuir ainda o risco de ser abordado este entregador passa a relação de delegacias da vizinhança e deseja não se distanciar muito dos quarteirões que possuem essas delegacias.

Para modelar este problema, utilizarei a criação de grafos, onde cada vértice é um quarteirão e cada aresta que liga estes vértices são as ruas. Este grafo não é direcionado, pois todas as ruas são mão dupla, ou seja, permitem se caminhar nos dois sentidos, além do grafo ser ponderado, onde cada aresta (rua) possui uma probabilidade de ser assaltado, este valor é o peso desta aresta.

Desejo através de um caminhamento neste grafo, retornar o seu caminho “mais curto”, ou seja, o caminho com menor probabilidade de ser assaltado, caso este caminho existir. Pois se este caminho não existir, deve ser retornado -1.

2. Modelagem e Solução

2.1. Estrutura de dados

2.1.1. Grafo (Matriz de adjacência)

A matriz foi implementada de acordo com o livro texto do Nívio Ziviani. A matriz de adjacência de um grafo de n vértices, possui ***n x n*** células, em que A[i][j] possui um valor diferente de 0, cujo existe uma aresta que sai do vértice i para o j, no nosso caso esse valor é referente a probabilidade de ser assaltado nessa rua. Além disso essa matriz é simétrica, pois o grafo é não direcionado.

O algoritmo abaixo demonstra como a matriz é criada e como é preenchida com seus valores de peso.

```
typedef struct TipoGrafo{
    TipoPeso Mat[MAXNUMVERTICES + 1][MAXNUMVERTICES + 1];
    int NumVertices;
    int NumArestas;
} TipoGrafo;

void FGVazio(TipoGrafo *Grafo){
    short i, j;
    for(i = 0; i <= Grafo->NumVertices; i++){
        for(j = 0; j <= Grafo->NumVertices; j++)
            Grafo->Mat[i][j] = 0;
    }
}

void InsereAresta(TipoValorVertice *V1, TipoValorVertice *V2, TipoPeso
*Peso, TipoGrafo *Grafo){
        Grafo->Mat[*V1][*V2] = *Peso;
}
```

2.1.2 Algoritmo de Dijkstra

Para resolução deste problema, após ter modelado meu problema em grafo, irei realizar um caminhamento nesse grafo, de forma a descobrir o caminho mais seguro, mais curto.

Este algoritmo utiliza uma técnica chamada de relaxamento, consiste em verificar se é possível melhorar o melhor caminho obtido até o momento até v se passarmos por u. Se isso acontecer, então p[v] e Antecessor[v] devem ser atualizados.

O algoritmo recebe como parâmetro de entrada o grafo a ser caminhado e a raiz, ou seja, qual vértice será iniciado o caminhamento. Assim cada vértice começa com seu p[i] como infinito, e o vértice raiz com seu valor igual a zero. No primeiro passo ele pega todos seus vizinhos adjacentes e setam o seus respectivos **p**, com o peso da sua aresta para ser alcançado a partir da raiz, e ao mesmo tempo coloca todos esses vizinhos adjacentes numa lista de prioridade (heap).

Depois cada item desse heap é analisado da mesma forma, e é verificado da mesma forma seus vizinhos adjacentes e o “preço” a ser pago para alcançar cada vértice através desse caminho, caso este vértice já tenha sido alcançado é verificado se esse novo caminho é mais barato que o escolhido anteriormente, caso seja, os valores devem ser atualizados. Essas operações acontecem até que o heap esteja vazio.

2.2 Módulos

2.2.1 tp1.c

Este módulo é responsável pela execução principal do programa, é nele que o arquivo de entrada é lido e o de saída gravado. Através do arquivo de entrada é possível gerar o grafo e para ser percorrido posteriormente pelo algoritmo de Dijkstra.

2.2.2 Grafo.c

Neste módulo que é feito toda a parte de execução do problema, onde fica os métodos para operar o grafo e a heap, também é onde está o algoritmo de Dijkstra. Assim a matriz de adjacência é gerada, utilizando os métodos FGVazio(TipoGrafo *Grafo), que é responsável por criar um grafo vazio, uma matriz vázia **nxn,** onde n é a quantidade de vértices do grafo, e para cada célula dessa matriz existe um valor que é referente ao peso da

aresta, ou zero, caso não exista aresta entre estes vértices. A inserção de cada aresta é feita utilizando o método InsereAresta, que seu trabalho simplesmente é setar a célula correspondente aquela aresta com seu respectivo valor de peso. As outras funções de operação com grafo são responsáveis para realizar o caminhamento dos vizinhos.

Os operadores para manter o heap são feitos de acordo com a implementação do livro do Nivio, e está heap é utilizada como uma lista de priodidade para o algoritmo de Dijkstra.

2.3 Análise de complexidade

2.3.1 FGVazio

A complexidade desse método é **O(v²),** pois é necessário criar a matriz vxv, onde v é a quantidade de vértices e setar cada célula com valor zero.

2.3.2 Insere Aresta

Esse método tem complexidade **O(1),** pois ele recebe como parâmetro os vértices que essa aresta liga e o peso da mesma, ou seja, é necessário apenas fazer uma atribuição. Mat[v1][v2] = peso;

2.3.3 Busca Adjacentes

Na matriz de adjacência os vizinhos de cada vértice é buscado verificando se existe uma aresta ligando estes vértices, ou seja, verifica se para cada célula da matriz o valor é diferente de zero, caso seja, estes dois elementos são vizinhos.

2.3.4 Dijkstra

Este algoritmo começa sua execução através da raiz, ou seja, um vértice de origem para ser calculado o menor caminho para todos os outros vértices do grafo. A cada passo, um vértice é adicionado à árvore S de caminhos mais curtos. Essa estratégia é gulosa, uma vez que a árvore é aumentada a cada passo com uma aresta que contribui com o mínimo possível para o custo total de cada caminho.

Como no nosso caso a lista de prioridade foi implementada utilizando o heap, podemos dizer que o procedimento constrói do programa tem complexidade **O(v).** O tempo total para executar a operação retira o item com menos peso é **O(v log v).** O while mais interno que percorre a lista de adjacência é executado **O(A).** A operação DiminuiChave é executada sobre o

heap A na posição Pos[v], a um custo **O(log V)**. Logo, o tempo total para executar o algoritmo de Dijkstra é O(V log V + A log V) = O(A log V)

3. Análise Experimental

- Problemas de execução. Vide Conclusão.

4. Especificações de máquina

- Intel Core 2 Quad Q8200 2,33 GHz
- 4 GB de ram ddr2
- Sistema operacional 32 bits, Windows 7.

5. Conclusão

Neste trabalho tive a oportunidade de testar os conhecimentos a cerca de estruturas do tipo grafo e realizar a resolução do problema em questão. Tive algumas dificuldades ao seguir o livro do Nívio, pois o mesmo apresenta alguns erros nos códigos dos algoritmos, não consegui descobrir o motivo exato do erro do código. Tive dificuldade de trabalhar com o relaxamento de arestas, para verificar o calculo exato do menor caminho, para cada vértice partindo de uma origem (Raiz).