TRABALHO PRÁTICO 1:
Grafos

Sandro Miccoli - 2009052409 - smiccoli@dcc.ufmg.br

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

18 de outubro de 2012

Resumo. *Esse relatório descreve como foi foi solucionado o problema proposto no Trabalho Prático 1, o qual será detalhado ao longo das próximas seções. Será descrito também a modelagem do problema e a solução proposta para tal. Finalmente será detalhado a análise de complexidade dos algoritmos, os testes utilizados para comprovar tais análises e uma breve conclusão do trabalho implementado.*

1. INTRODUÇÃO

A situação que deparamos neste trabalho é a seguinte: a distribuidora de produtos, Atlanticon, está precisando de uma consultoria para decidir em qual cidade deve instalar a nova filial para servir uma determinada região.

A empresa possui conhecimento das distâncias entre quaisquer duas cidades da região que sejam conectadas por algum trilho. Além disso, o trem, precariamente, realiza a entrega de apenas um produto por viagem, tornando necessário retornar para a filial para carregar um novo produto.

Existem 3 cenários distintos que a Atlanticon deseja saber qual a melhor opção de cidade para instalar a nova filial, e o objetivo do trabalho é solucionar esse problema.

Descreveremos aqui os três cenários:

Cenário 1: Escolher a cidade de modo a minimizar os custos de gasto em combustível. Considerando que cada cidade emite a mesma quantidade de pedidos em um mesmo intervalo de tempo.

Cenário 2: Escolher a cidade para também minimizar os custos de gasto em combustível. Porém, neste cenário cada cidade possui um volume diferente de pedidos, isso deve ser considerado para a escolha final.

Cenário 3: A Atlanticon deseja garantir que o produto seja entregue num prazo de X horas. A escolha da cidade se dá pelo menor valor possível de X que pudermos encontrar.

O restante deste relatório é organizado da seguinte forma. A Seção 2 descreve como foi feita a modelagem e manipulação dos grafos. A Seção 3 descreve rapidamente qual foi a solução proposta além do método utilizado para calcular as distâncias mínimas entre as cidades. A Seção 4 trata de detalhes específicos da implementação do trabalho: quais os arquivos utilizados; como é feita a compilação e execução; além de detalhar o formato dos arquivos de entrada e saída. A Seção 5 contém a avaliação experimental, quantificando o tempo de execução de cada operação com grafos de diversos tamanhos e formatos. A Seção 6 conclui o trabalho.

2. MODELAGEM

Inicialmente, para trabalhar com grafos, foi criada uma estrutura que contém a informação de quantos vértices o grafo possui, nesse caso, cidades, e qual o volume de pedidos de cada cidade:

```
typedef struct grafo {
    Matriz matrizAdj;
    int N; // Quantidade de cidades no grafo
    int *Vol; // Volume de pedidos de cada cidade
} Grafo;

typedef struct Matriz{
    int col, lin;
    int ** matriz;
} Matriz;
```

Já que o grafo que nos é passado como entrada é uma matriz de adjacências, e como já havíamos implementado uma estrutura de matriz no Trabalho Prático 0, reutilizamos o código neste trabalho.

A complexidade de espaço dessa estrutura pode ser considerada como $O(n)$, sendo n o tamanho bidimensional da matriz, ou seja, o produto entre a quantidade de linhas e colunas que ela possui.

Como foi detalhado na especificação, a matriz foi modelada para ser alocada e desalocada dinâmicamente, através dos comandos *malloc* e *free*. Para confirmar que este processo de alocação dinâmica de memória estava ocorrendo como esperado, foi utilizado o comando *valgrind* para verificar qualquer tipo de vazamento de memória.

3. SOLUÇÃO PROPOSTA

O problema que deparamos é minimizar o custo de combustível dos trens da empresa Atlanticon. Como já modelamos nosso problema como um grafo, nada melhor que um algoritmo específico que nos traga a solução. A seguir uma definição de [Sedgewick] para o problema de caminho mínimo:

> Dado um digrafo com custos não negativos nos arcos e um vértice *s*, encontrar uma arborescência de caminhos mínimos com raiz *s* no digrafo.

O algoritmo que utilizaremos foi inventado por Edsger Dijkstra [a pronúncia é algo entre ”Dêcstra” e ”Dêicstra”] e publicado em 1959. O algoritmo pode ser usado, em particular, para encontrar um caminho de custo mínimo de um dado vértice a outro [Sedgewick].

O algoritmo de Dijkstra assemelha-se ao busca em largura, mas é um algoritmo guloso, ou seja, toma a decisão que parece ótima no momento. Um exemplo prático do problema que pode ser resolvido pelo algoritmo de Dijkstra é: alguém precisa se deslocar de uma cidade para outra. Para isso, ela dispõe de várias estradas, que passam por diversas cidades. Qual delas oferece uma trajetória de menor caminho? [Wikipedia]

A solução proposta aqui foi realizar o algoritmo de Dijkstra para cada vértice do grafo, então descobrir o menor caminho dentre todos os possíveis para resolver nosso problema de distribuição de produtos pelas linhas de trem.

Para o **Cenário 1** utilizamos apenas as distâncias entre as cidades como peso nas arestas para calcular o caminho mínimo entre os vértices.

Já no **Cenário 2**, para cada cidade, calculamos o produto entre a distância das cidades com o volume de pedidos de cada uma. O menor produto resulta na nossa melhor escolha.

No **Cenário 3**, temos que minimizar o valor de X, para isso inicialmente calculamos a maior distância percorrida para cada cidade. Assim garantimos a quantidade de horas mínima para o pedido chegar. Dentre essas maiores distâncias, escolhemos a menor delas, minimizando o valor de X.

Além desses cenários, tempos que retornar qual o prejuízo percentual acarretado quando a cidade é escolhida, de modo a garantir o menor tempo de entrega máximo. Para isso calculamos a razão entre os custos do **Cenário 1** e **Cenário 3**. Isso nos retorna qual a porcentagem de aumento de combustível caso a Atlanticon escolher a cidade do **Cenário 3** para implantar a filial.

3.1. Principais funções implementadas

3.1.1. Dijkstra

- *void dijkstra (Grafo G, int v, int *dis)*

  Descrição: Soluciona o problema do caminho mais curto para o vértice v e armazena o resultado no vetor de distâncias dis.
  Parâmetros: Estrutura de grafo G, vértice v e vetor de distâncias dis.
  Complexidade: $O(n^2)$, onde n são os vértices do grafo.

- *void dijkstra_all (Grafo G)*

  Descrição: Soluciona o problema do caminho mais curto para todos os vértices do grafo e atualiza a matriz de adjacência contida no grafo transformando-a em uma matriz de distâncias mínimas.
  Parâmetros: Estrutura de grafo G.
  Complexidade: $O(n^3)$, onde n são os vértices do grafo.

3.1.2. Consultoria

- *int cenarioUm(Grafo G)*

  Descrição: Pesquisa no grafo G, que agora contém uma matriz de distâncias mínimas, para descobrir qual cidade possui o menor caminho a ser percorrido pelo grafo. Retorna o índice da melhor cidade.
  Parâmetros: Estrutura de grafo G.
  Complexidade: $O(n^2)$, onde n são os vértices do grafo.

- *int cenarioDois(Grafo G)*

  Descrição: Vasculha o grafo G, que agora contém uma matriz de distâncias mínimas, para descobrir qual cidade possui o menor caminho a ser percorrido pelo grafo. Porém, agora considerando o volume médio de pedidos de cada cidade. Retorna o índice da melhor cidade.
  Parâmetros: Estrutura de grafo G.
  Complexidade: $O(n^2)$, onde n são os vértices do grafo.

- *int cenarioTres(Grafo G)*

  Descrição: Caminha pelo grafo G, que agora contém uma matriz de distâncias mínimas, para descobrir qual o menor caminho dentre os maiores caminhos percorridos de cada cidade. Retorna o índice da melhor cidade.
  Parâmetros: Estrutura de grafo G.
  Complexidade: $O(n^2)$, onde n são os vértices do grafo.

- *float prejuizoCen3(Grafo G, int cen1, int cen3)*

  Descrição: Calcula o custo do cenário 1 para a cidade de índice $cen1$ e do cenário 3 para a cidade de índice $cen3$. Depois é calculado a razão entre esses custos e esse é o prejuízo de se escolher o cenário 3 em vez do 1. O valor retornado é um $float$ contendo o prejuízo em porcentagem.
  Parâmetros: Estrutura de grafo G, índice da cidade do cenário 1 $cen1$, índice da cidade do cenário 3 $cen3$.
  Complexidade: $O(n)$, onde n são os vértices do grafo.

4. IMPLEMENTAÇÃO

4.1. Código

4.1.1. Arquivos .c

- **tp1.c** Arquivo principal do programa, lê todas as instâncias de problemas do arquivo de entrada, realiza os cálculos dos cenários e insere cada resultado no arquivo de saída.
- **matriz.c** Contém todas as funções de manipulação, leitura e escrita de matrizes.
- **grafos_matriz.c** Contém todas as funções de manipulação, leitura e escrita de grafos. Utiliza o TAD de matriz como implementação da matriz de adjacências.
- **dijkstra.c** Contém a implementação do algoritmo de Dijkstra, tanto para um vértice quanto para todos os vértices do grafo.
- **consultoria.c** Contém a implementação de todos os cenários que a Atlanticon estabeleceu.

4.1.2. Arquivos .h

- **matriz.h** Além de definir a estrutura de matriz, contém o cabeçalho todas as funções de manipulação, leitura e escrita de matrizes.

- **grafos_matriz.h** Além de definir a estrutura de grafos, contém todas as funções de manipulação, leitura e escrita de grafos. Utiliza o TAD de matriz como implementação da matriz de adjacências.
- **dijkstra.h** Contém o cabeçalho do algoritmo de Dijkstra, tanto para um vértice quanto para todos os vértices do grafo.
- **consultoria.h** Contém o cabeçalho de todos os cenários que a Atlanticon estabeleceu.

4.2. Compilação

O programa deve ser compilado através do compilador GCC através de um makefile ou do seguinte comando:

```
gcc -Wall -Lsrc src/tp1.c src/arquivos.c src/grafos_matriz.c
                src/matriz.c src/dijkstra.c src/consultoria.c -o tp1
```

Ou através do comando $make$.

4.3. Execução

A execução do programa tem como parâmetros:

- Um arquivo de entrada contendo várias instâncias de grafos e seus respectivos volumes de pedidos pra cada cidade.
- Um arquivo de saída que irá receber o resultado dos cálculos de cada cenário e o percentual de prejuízo, já explicado na Seção 3.

O comando para a execução do programa é da forma:

```
./tp1 <arquivo_de_entrada> <arquivo_de_saída>
```

4.3.1. Formato da entrada

A primeira linha do arquivo de entrada contém o valor *k* de instâncias que o arquivo contém. A próxima linha contém a quantidade *n* de cidades que o grafo contém. As próximas *m* linhas contém a matriz de adjacência referente à esse grafo. Por último, a última linha contém *n* inteiros que são referentes ao volume médio de pedidos de cada cidade da região.

A Figura 1 contém um grafo com 7 cidades, esparso. Geramos um arquivo que contém a matriz de adjacência dele e o volume de pedidos de cada cidade. Esse arquivo de entrada tem a seguinte configuração:

```
1
7
0 4 2 0 4 0 1
4 0 5 0 7 0 0
2 5 0 0 4 8 6
0 0 0 0 3 0 0
4 7 4 3 0 7 0
0 0 8 0 7 0 0
1 0 6 0 0 0 0
217 730 272 684 339 890 356
```

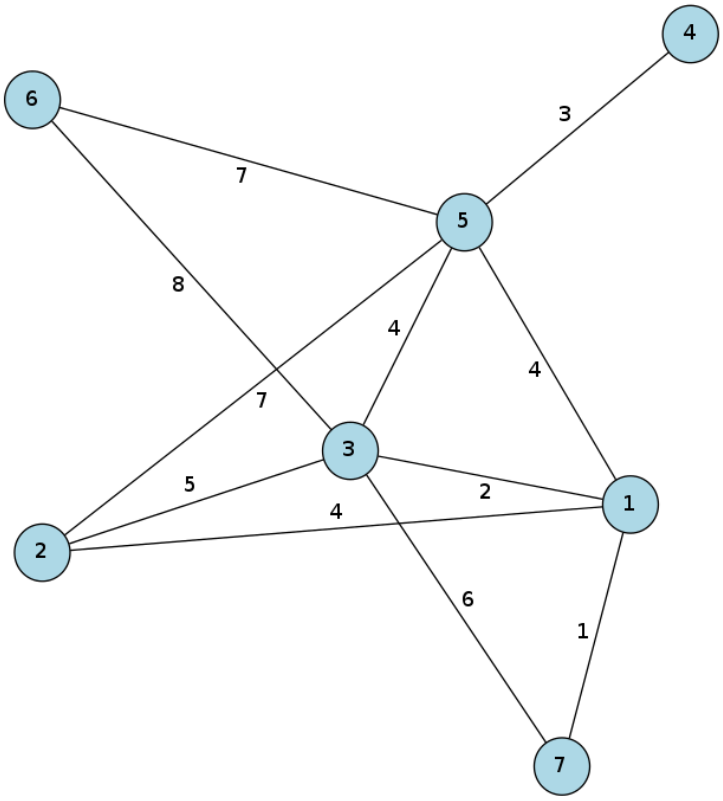

Figura 1. Grafo esparso com 7 cidades

4.3.2. Formato da saída

O arquivo de saída contém 3 inteiros e um valor de ponto flutuante. Cada inteiro é um identificador da cidade na qual a filial deve ser instalado considerando cada um dos cenários descritos na Seção 3. O último valor consiste no aumento percentual ao se escolher a cidade de modo a minimizar o tempo máximo de entrega ao invés de minimizar o gasto total de combustível.

Considerando o grafo da Figura 1, o resultado obtido após a execução do programa é escrito no arquivo de saída, que tem a seguinte configuração:

```
1 5 5 7.14
```

5. AVALIAÇÃO EXPERIMENTAL

Para testar o algoritmo realizamos quatro testes, separados em dois grupos. O primeiro grupo de testes se refere à grafos esparsos e o segundo a grafos densos.

Para cada grupo de testes fizemos um teste que varia a quantidade instâncias, com grafos de tamanho fixo (25 vértices), e outro teste com apenas uma instância e grafos de tamanhos crescentes.

5.1. Máquina utilizada

Segue especificação da máquina utilizada para os testes:

```
model name:     Intel(R) Core(TM) i3 CPU       M 330  @ 2.13GHz
cpu MHz:        933.000
cache size:     3072 KB
MemTotal:       3980124 kB
```

5.2. Grafos Esparsos - Variando quantidade de instâncias

Na Figura 2, é possível ver o tempo de execução para o primeiro teste. Aqui fixamos o tamanho do grafo para 25 vértices e variamos o número de instâncias, de 1 até 500. O maior tempo foi de 34 segundos, para 500 instâncias.

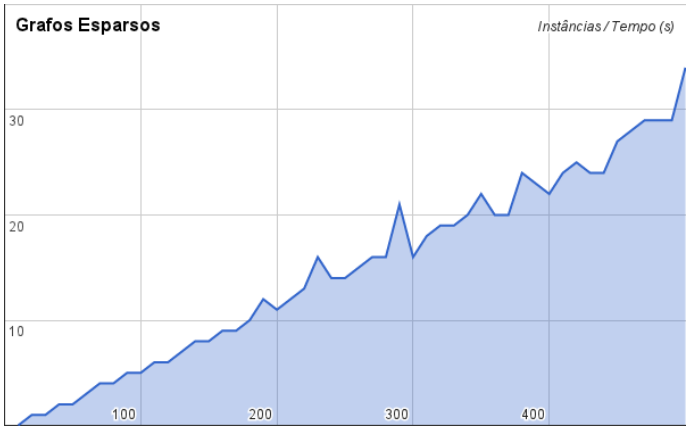

Figura 2. Grafos esparsos com instâncias variando até 500

5.3. Grafos Esparsos - Variando tamanho dos grafos

Na Figura 3, é possível ver o tempo de execução para o segundo teste. Aqui fixamos quantidade de instâncias para apenas uma, e variamos a quantidade de vértices do grafo, que foi de 1 até 500. O maior tempo de execução foi de 246 segundos, para 500 vértices.

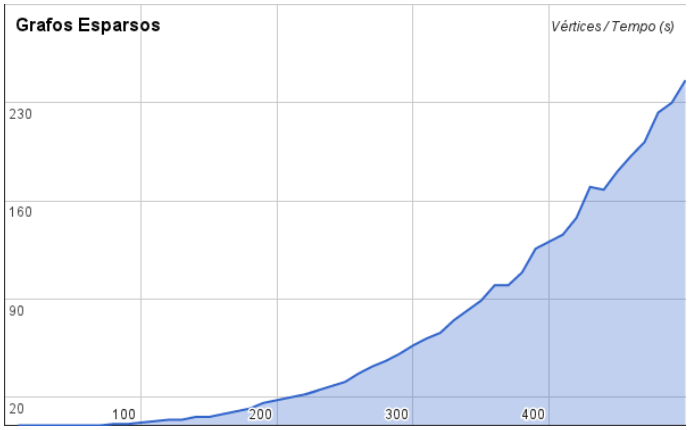

Figura 3. Grafos esparsos com grafos variando até 500 vértices

5.4. Grafos Densos - Variando quantidade de instâncias

Na Figura 4, é possível ver o tempo de execução para o terceiro teste. Aqui fixamos o tamanho do grafo para 25 vértices e variamos o número de instâncias, de 1 até 500. O maior tempo foi de 30 segundos, para 500 instâncias.

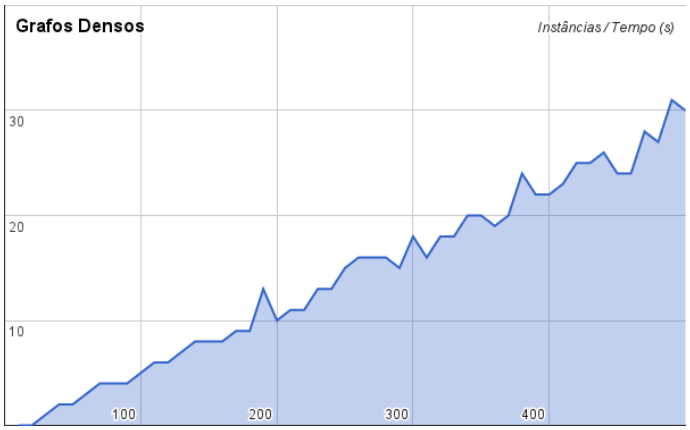

Figura 4. Grafos densos com instâncias variando até 500

5.5. Grafos Densos - Variando tamanho dos grafos

Na Figura 5, é possível ver o tempo de execução para o quarto e último teste. Aqui fixamos quantidade de instâncias para apenas uma, e variamos a quantidade de vértices do grafo, que foi de 1 até 500. O maior tempo de execução foi de 224 segundos, para 500 vértices.

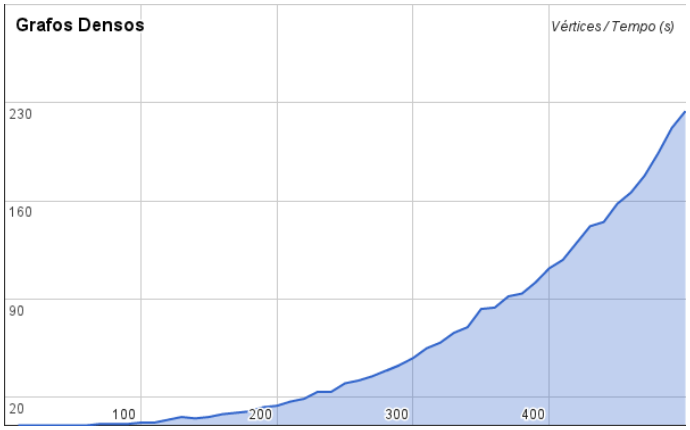

Figura 5. Grafos densos com grafos variando até 500 vértices

5.6. Resultado

O resultado geral dos testes ocorreu como esperado. Os testes em que foi variada apenas as instâncias (5.2 e 5.4) cresceu de forma esperada, em um crescimento quase que constante.

Já os testes em que variamos os tamanhos dos grafos (5.3 e 5.5) já deu pra visualizar melhor a complexidade do algoritmo de Dijkstra. Pelos gráficos é perceptível o crescimento exponencial de cada teste.

Outra conclusão interessante é perceber que o Dijkstra que foi implementado neste trabalho funciona melhor para grafos densos do que para grafos mais esparsos. Veja a Figura 6 para uma melhor comparação entre os testes 5.3 e 5.5. Apesar da diferença não ser muito significativa, para entradas maiores a diferença só tende a crescer.

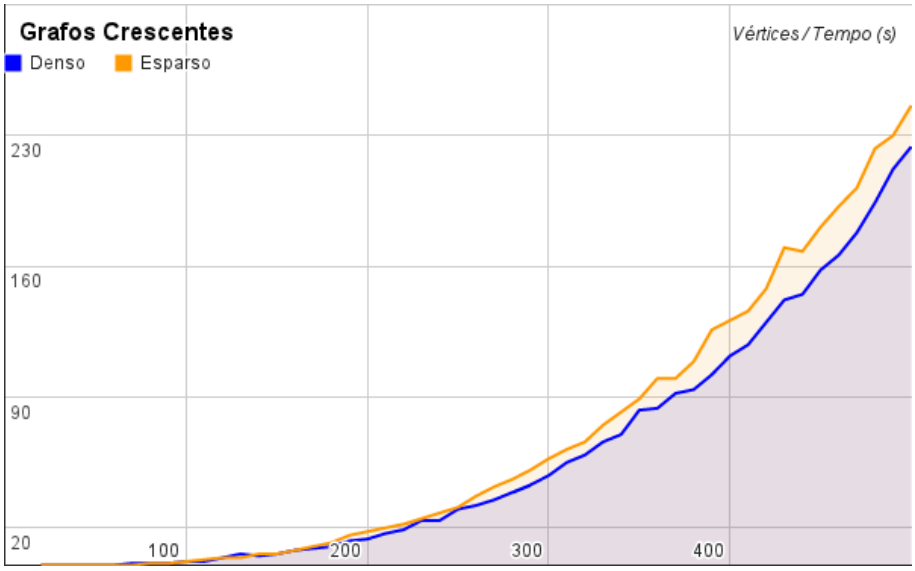

Figura 6. Grafos densos e esparsos com grafos variando até 500 vértices

6. CONCLUSÃO

Modelamos o problema das linhas de trem e gasto com combustível com grafos, para assim, podermos utilizar algoritmos específicos que encontrassem o menor caminho de uma cidade até outra. Esta etapa do trabalho foi bem satisfatória. Pode-se dizer que a consultoria prestada para a distribuidora Atlanticon será vista com bons olhos, pois consegue solucionar o problema de forma eficiente e elegante.

Vimos que o algoritmo de Dijkstra funciona melhor para grafos densos do que para grafos mais esparsos, o que na prática é o que acontece nos mapas reais. As cidades estão todas bem conectadas, porém cabe ao motorista decidir qual o melhor caminho. Neste caso, coube ao algoritmo definir, para cada cenário, qual seria a melhor opção de caminho, e, assim, escolher a melhor cidade para instalar a nova filial.

Referências

Sedgewick, R. *Algorithms in C: Graph Algorithms*.

Wikipedia. Algoritmo de dijkstra. `http://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra`.