UFMG - UNIVERSIDADE FEDERAL

DE MINAS GERAIS

AEDS 3 -TRABALHO PRÁTICO 2

Nome: João Pedro Samarino

Matricula:2013048933

Belo Horizonte

2014

1. Introdução

O trabalho teve como objetivo a implementação de um programa que dada uma cidade fictícia, onde cada rua é avenida possuem o mesmo tamanho, o mesmo tem que achar a quantidade de menores caminhos entre uma esquina pré-definida a partir de um ponto inicial localizado na primeira esquina da diagonal esquerda superior, podendo ainda existir bloqueios em N esquinas da cidade. Para realizar este trabalho foi necessário aprender um pouco de paradigmas de programação como divisão e conquista, programação dinâmica, deste modo aplicando alguns conceitos aprendidos em sala de aula.

Os problemas combinatórios são bem comuns na área das ciências da computação, este problema em questão pode ser classificado nesta categoria, pois existe N caminhos de um ponto ao outro e estes são combinações das ligações de uma esquina a outra. O algoritmo usado deve contar os caminhos que possuem a menor distancia possível.

Para achar os caminhos possíveis o algoritmo usado cria uma tabela de custo onde à mesma representa todas as esquinas, pois abstraindo este problema mais profundamente vemos que as ruas e avenidas que ligam as esquinas são irrelevantes para o problema, pois o importante é saber para qual esquina o algoritmo deve seguir para achar um menor caminho possível, esta tabela possui valore em cada esquina onde os menores mostram a menor distancia possível até o ponto de partida, depois o algoritmo de busca de caminhos conta a quantidade de caminhos para obter a solução proposta do trabalho.

2. Solução Proposta

A solução proposta constitui de simples algoritmos para processamento do texto de entrada, modularização da matriz de custo e outro para contar a quantidade de menores caminhos na matriz de custo. Para facilitar o trabalho foi criado um tipo abstrato de dados, e o programa foi separado em funções (modulos) elementares independentes, desta maneira foi possível criar um programa simples é menor.

Apesar da solução do trabalho ser simples, para se ter um bom aproveitamento dos recursos da linguagem, primeiro foi necessário pensar em uma estrutura de dados que iria atender, alem da organização das funções, equilibrando desempenho e simplicidade.

A idéia por trás da solução deste problema é não tentar achar todas as combinações de esquinas possíveis entre os pontos iniciais e finais, pois este tipo de solução tem um custo computacional muito elevado. Abstraindo mais profundamente o problema percebemos que existem subproblemas nesta estrutura, estes são representados por uma tabela de custo, a qual mostra o custo de pegar um determinado caminho dado às direções disponíveis possíveis, então um algoritmo de busca recursiva conta às direções que tem o mesmo valor sempre, chegando deste modo ao ponto desejado e contando a quantidade de caminhos que ele percorreu, achando assim sempre a quantidade de menores caminhos. A explicação detalhada em relação como chegar a esta tabela de custo e sobre os algoritmos utilizados estão localizadas nas seções a seguir deste trabalho.

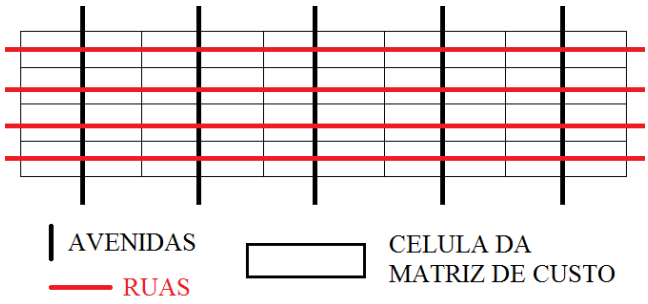
2.1. Estrutura de Dados:

Para entendermos o funcionamento do programa primeiro temos que entender a estrutura de dado abstrata criada para a solução, para então explorarmos os algoritmos de maneira completa. Foi criada estrutura básica para armazenamento de registros de maneira mais abstrata, ou seja, para facilitar a manipulação pelas funções dos registros lá armazenados. As estruturas de dados estão contidas no arquivo (caminhos.h).

As estruturas de dados abstratos são:

- typedef typedef struct cidade_{ int** matriz_custo; int qt_linhas; int qt_colunas; } cidade;

A estrutura (cidade) é responsável por armazenar apontadores de apontadores do tipo inteiro, estes apontadores iram apontar para a matriz de custo que é alocada dinamicamente durante a execução do programa. Esta estrutura também possui duas variáveis inteiras que representam (qt_linhas) quantidades de ruas, (qt_colunas) quantidade de avenidas, em uma instancia do problema, também representam a dimensão da matriz de custo, pois a mesma representa as esquinas da cidade. Abaixo podemos ver um exemplo desenhado da representação da matriz de custo vazia onde cada espaço da matriz representa um cruzamento, podendo desta forma representar todo o problema usando apenas esta matriz, lembrando que cada espaço é preenchido nas funções a seguir, tendo no final das operações o valor do custo para chegar ao ponto inicial.

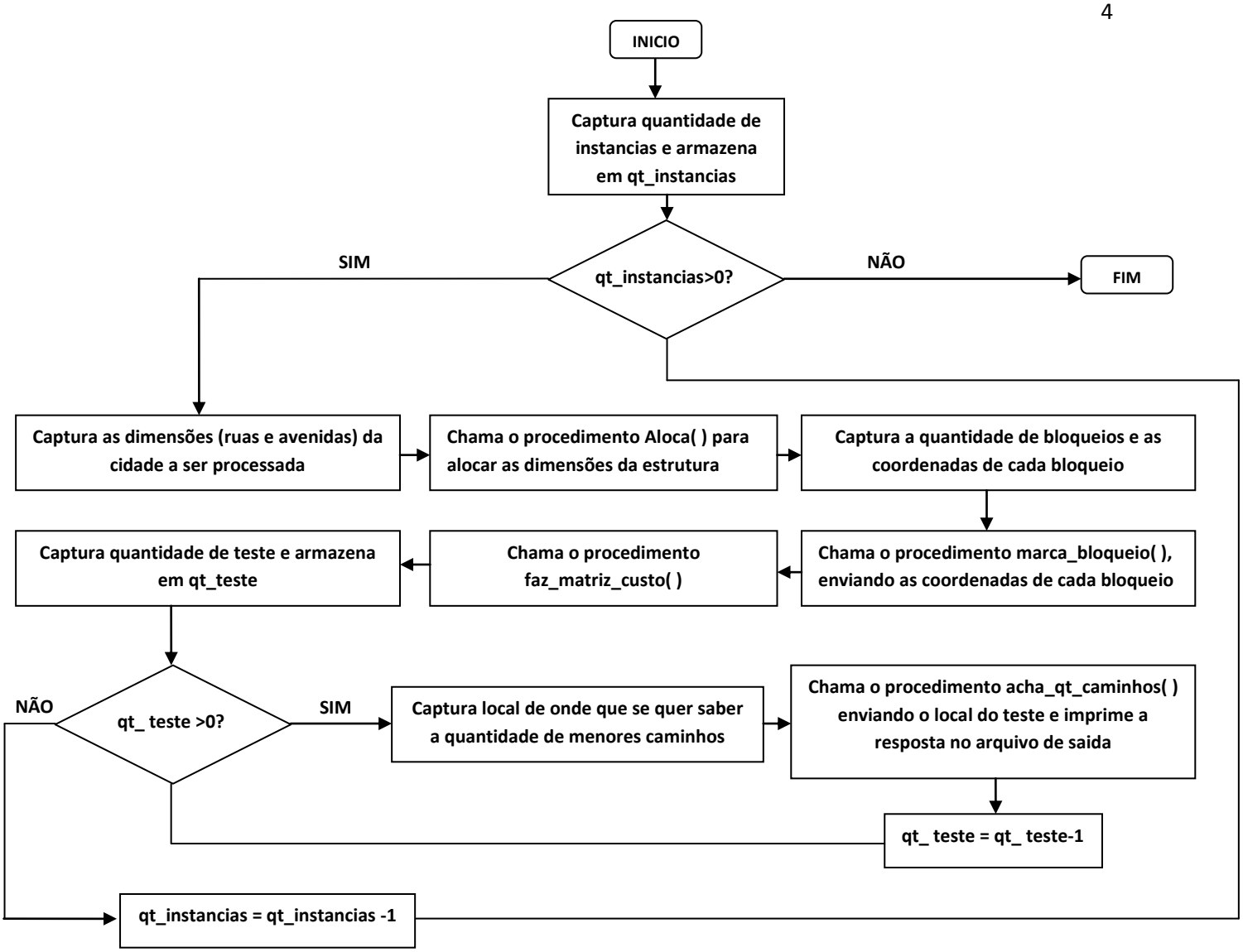


2.2.Funções e Procedimentos

No programa foram criados procedimentos para manipulação de dados é execução da proposta de solução que tem como objetivo gerar o arquivo solução. O arquivo (main.c) contem procedimentos do fluxo principal e o (caminhos.c) procedimentos relacionados ao TAD.

Funções e procedimentos do arquivo (main.c) :

*Função Principal (int main(int argc, char *argv[])):* A função “main” compõem o fluxo principal do programa , deste modo tem como objetivo principal capturar informações e realiza todas as chamadas ao TAD quando necessário , por fim retornar a quantidade de caminhos de cada operação para o arquivo texto. O fluxograma a baixo da função “main” mostra como funciona a lógica de execução do programa.



Funções e procedimentos do arquivo (caminhos.c) :

- *void Aloca (cidade *c):* O procedimento tem como objetivo alocar a matriz de custo dentro da estrutura (cidade), recebida pelo procedimento, a quantidade de linhas e colunas são delimitadas pelas variáveis (qt_coluna) e (qt_linha) que estão internas a estrutura (c). Para gerar a matriz na estrutura o programa primeiro aloca um vetor de ponteiros correspondente as linhas da matriz e depois através de um loop que e dado pela quantidade de linhas ele aloca as colunas. No final do procedimento a matriz é preenchida com (-2), este é um marcador que sinaliza que a célula esta vazia, Na posição (0,0) desta matriz o procedimento coloca o valor de (0) para sinalizar que está é a posição do inicio do percurso. Abaixo podemos ver um exemplo do preenchimento da matriz de custo em uma matriz que representa 4 ruas e 5 avenidas , ou seja , qt_coluna = 5 e qt_linha=4.

0	-2	-2	-2	-2
-2	-2	-2	-2	-2
-2	-2	-2	-2	-2
-2	-2	-2	-2	-2

LEGENDAS:

-2 Espaço vazio

0 Ponto inicial

- *void marca_bloqueio(cidade *c,int linha , int coluna):* O objetivo deste procedimento é simplesmente marcar os cruzamentos bloqueados na matriz de custo da estrutura (c) , ele marca na matriz de custo na linha informada pela variável (linha) e na coluna informado por (coluna), ele coloca o marcador (-1) nesta posição. Pegando o exemplo acima e colocando os bloqueios em linha = 2 e coluna = 2, a nova matriz de custo pode ser vista abaixo.

0	-2	-2	-2	-2
-2	-2	-2	-2	-2
-2	-2	-1	-2	-2
-2	-2	-2	-2	-2

LEGENDAS:

-2 Espaço vazio

0 Ponto inicial

-1 Cruzamento bloqueado

- *void faz_matriz_custo(cidade *c, int linha, int coluna, int valor):* Esse procedimento é o mais complexo do programa , pois o mesmo preenche as distancias na matriz de custo, para fazer isso é necessário que o mesmo faça chamadas recursivas nas quatro direções em relação sua posição atual, dada pelas variáveis (linha) e (coluna). Inicialmente o programa verifica as quatro posições a sua volta, nestas as que têm (-2) ou possuem um valor maior que ((valor)+1), o procedimento coloca nesses espaços o numero representado por ((valor)+1), agora se algum espaço a sua volta possuir uma valor menor que ((valor)-1) a função se encerra imediatamente , este valor mostra que o caminho percorrido não é o menor possível. Após colocar os valores nos espaços possíveis a sua volta este procedimento faz uma chamada recursiva nas direções as quais foram preenchidas, enviando no cabeçalho da chamada as informações desta nova célula. Este procedimento por ser recursivo e sua condição de parada ser as citadas acima garante que o mesmo ache todos os caminhos do percurso desejado. Vale lembrar que a primeira chamada realizada tem que possuir os seguintes parâmetro (linha =0), (coluna =0), (valor =0), que é referente a célula inicial do caminho. Abaixo na primeira imagem podemos ver o que acontece na primeira chamada deste procedimento e a segunda imagem mostra a matriz de custo dos exemplos totalmente preenchida.

0 →	1	-2	-2	-2
1 ↓	-2	-2	-2	-2
-2	-2	-1	-2	-2
-2	-2	-2	-2	-2

LEGENDAS:

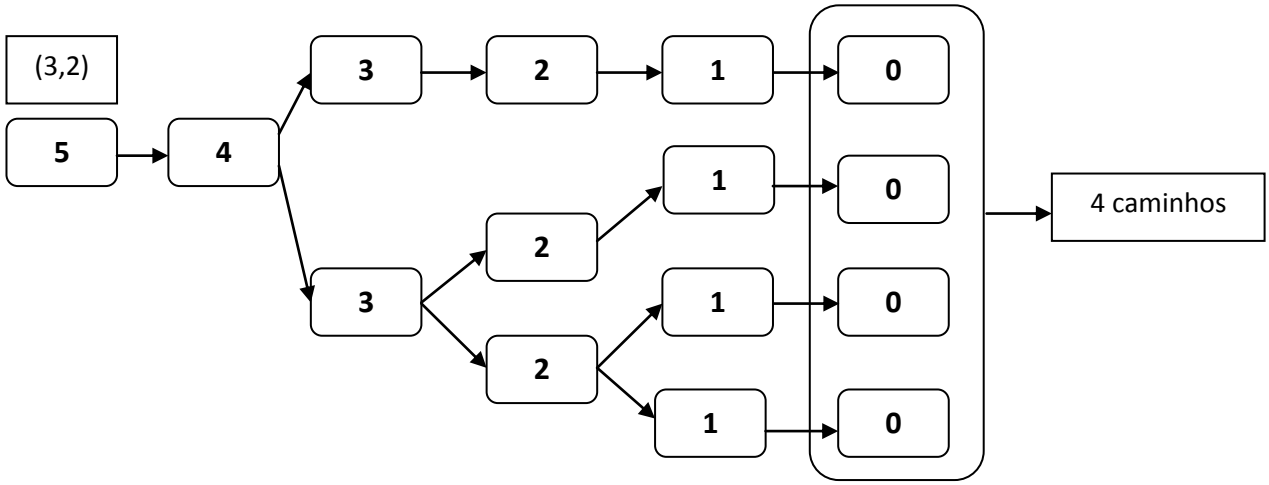
(1) - Valor preenchido pela primeira chamada atual

→ Chamadas recursivas executadas na chamada atual

MATRIZ CUSTO PREENCHIDA APOS O TERMINO DO PROCEDIMENTO

0	1	2	3	4
1	2	3	4	5
2	3	-1	5	6
3	4	5	6	7

- *void acha_qt_caminhos(cidade *c,int linha,int coluna,int *qt_caminhos):* Esse procedimento tem extrema importância para o programa , pois o mesmo dado a matriz de custo já preenchida busca os menores caminhos para a percorrer se dividindo recursivamente quando existe mais de uma opção de caminho. Para percorrer a matriz de custo o procedimento primeiro recebe a (linha) e (coluna) iniciais do processo, a partir dessas coordenadas o mesmo olha para as quatro direções possíveis e verificam quais possuem os menores valores que sejam maiores ou iguais a (0), então este faz uma nova chamada recursiva a elas e estas fazem a mesma coisa até que atinjam o valor de (0) ou não seja mais possível andar em direção alguma. Quando se atinge (0) a função incrementa o valor de (1) a variável (qt_caminhos) apontada nesse procedimento. Abaixo podemos ver a arvore que a função recursiva gera para o exemplo da explicação anterior, querendo saber o menor caminho a partir da posição (3,2) da matriz de custo.



*void limpa_estrutura(cidade *c*): Esse procedimento simplesmente tem como objetivo de limpar as estruturas alocadas para serem novamente usadas em outras instancias ou para a finalização do programa.

3.Análise de Complexidade

Procedimento Aloca: O procedimento tem uma complexidade de tempo O(n*m) onde n é o numero de linhas da matriz a ser alocada e m representa as colunas, esta complexidade se da devido aos loops que percorrem toda a estrutura. A complexidade de espaço do programa é O(n*m), pois esse procedimento tem como objetivo alocar uma estrutura em forma de matriz através da função malloc().

Procedimento marca_bloqueio: O procedimento tem uma complexidade de tempo e espaço é O(1), pois faz somente uma operação de atribuição.

Procedimento faz_matriz_custo: O procedimento tem uma complexidade no pior caso $O((n*m)^2)$, onde n e m se referem consecutivamente a quantidades de linhas e colunas da matriz resposta, está complexidade no pior caso é divido a possibilidade de cada (n) células da matriz chegarem a fazer (n) chamadas, já a complexidade no melhor caso é O(n*m) onde o procedimento marca todos os valores corretamente. A complexidade de espaço é O(1) pois são geradas neste procedimento somente variáveis de buffer.

Procedimento acha_qt_caminhos e limpa_estrutura: Os procedimentos tem uma complexidade de tempo O(n*m), onde n e m se referem consecutivamente a quantidades de linhas e colunas da matriz de custo, esta complexidade é divido ao fato de que a função pode fazer no máximo (m*n) chamadas recursivas pois o numero de menores caminhos é limitado ao numero de células.

A complexidade de espaço é O(1) pois são geradas neste procedimento somente variáveis de buffer.

Função main – função principal: a função principal faz 1 varreduras que podem varia de 0 a X , onde X é a quantidade de instancias informada no arquivo de entrada, no final de cada ciclo o programa chama as funções da TAD , porem ele pode chamar ate X vezes uma função $O((n*m)^2)$ alem de executar comando O(1) . Temos no pior caso $O(x)*O((n*m)^2)+O(1)=O((n*m)^2*x)$. Porem essa analise só pode ser feita considerando que todas as matrizes (cidades) da instancia vão ter o mesmo tamanho mantendo assim a relação “n”,”m” é sempre igual, se o tamanho das matrizes varia a complexidade de tempo pode variar também.A complexidade de espaço é O(n*m), o programa pode chamar X vezes uma função de complexidade O(n*m) porem a cada novo ciclo o espaço alocado e limpado para poder ser novamente alocado com um novo tamanho.

4.Implementação

Para desenvolver este trabalho, o mesmo foi dividido em três arquivos com o intuito de melhorar a organização e normalizar as funções e estruturas de acordo com os seus objetivos.

4.1.Arquivos utilizados

(main.c): Arquivo principal, responsável pelo fluxo do programa, contem funções relacionadas a entrada e saída de dados dos arquivos, contem a função (main()), que controla todas as chamadas ao TAD.

(caminhos.c): Arquivo responsável pelos procedimentos relacionados ao TAD, ou seja, aqueles que iram fazer operações diretamente com a estrutura de dados relacionada.

(caminhos.h): Este arquivo contem o cabeçalho das funções relacionadas ao TAD que estão contidas no arquivo (matriz.c), Também possui as estruturas de dados ligadas ao programa e suas definições.

4.2. Compilação

O programa deve ser compilado no compilador GCC através de um makefile ou do seguinte comando no terminal:

gcc main.c caminhos.c -o tp2

4.3. Execução

A execução do programa tem como parâmetros:

• Arquivo de entrada.

• Arquivo de saída.

O comando no terminal para a execução do programa e da forma:

./tp2 < arquivo de entrada > < Arquivo de saída >

4.3.1. formato dos arquivos

Arquivo de entrada: O arquivo de entrada deve conter as especificações básicas dos procedimentos que iram ser realizados pelo programa. Este ira conter na seguinte ordem formatada, (Instancias), (dimensão da cidade(1)), (números de bloqueios da cidade(1)),(localização dos bloqueios), (numero de testes) ,(localização dos lugares testados) e assim sucessivamente para cada instancia.

5. Experimentos

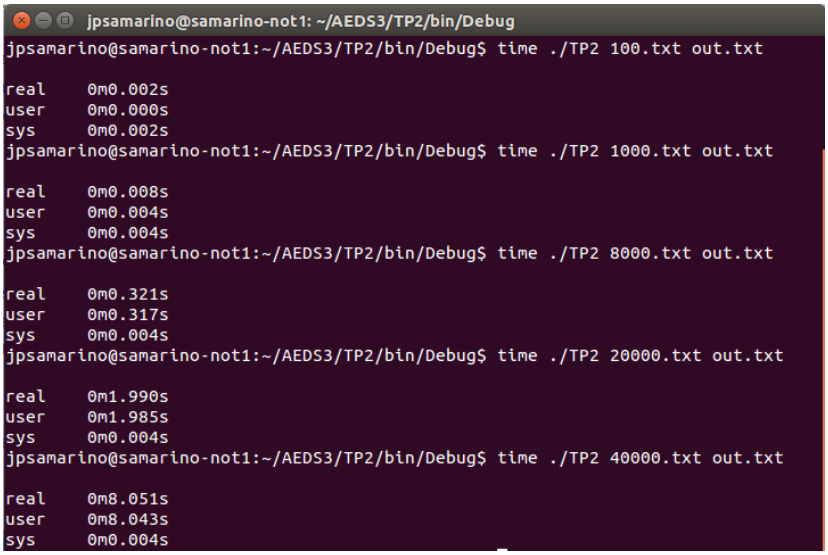
Os experimentos realizados tiveram como objetivo verificar o funcionamento do programa é o tempo gasto para execução do mesmo, verificando assim a qualidade dos algoritmos implementados no mesmo.

5.1. Maquina utilizada

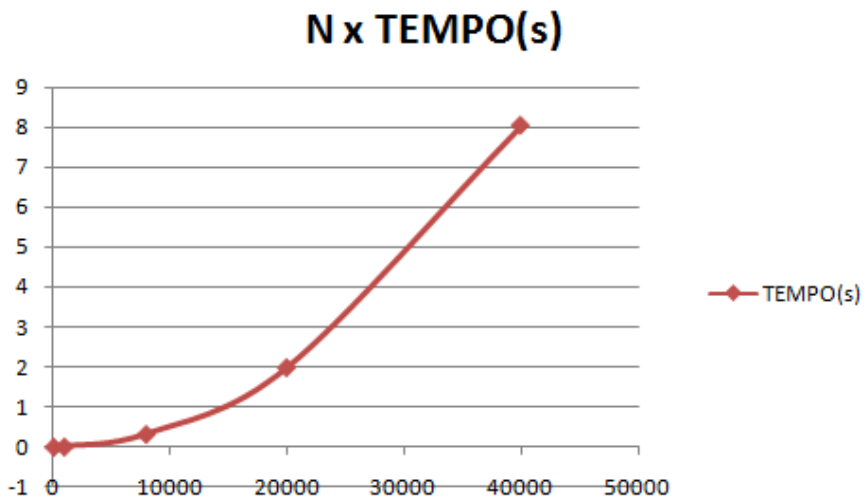
Os testes e a compilação do programa foram realizadas em um Pentium core 2 duo, com 3 Gb de memória principal, o sistema operacional utilizado foi o Ubuntu Linux para arquitetura AMD64 (64 bits por ciclo de maquina) .

5.2. Teste

Para verificar o desempenho do programa foram realizados alguns testes. Os arquivos de testes contem uma instância com uma cidade de (N) igual a consecutivamente 100, 1000, 8000, 20000, 40000, sendo (N) a quantidade de cruzamentos, ou seja, (m*n), realizando somente (1) teste por arquivo. Foi utilizado o comando time para medir o tempo de execução do programa para a entrada determinada. Podemos ver abaixo na imagem o tempo de execução para cada entrada.



Podemos notar que com o aumento de (N) o incremento do tempo não foi linear, pois como dissemos na analise de complexidade a mesma representa um polinômio que pode chegar a ser de grau (2). Abaixo existe um gráfico de (N x Tempo) onde podemos mostrar a relação entre o tempo de execução é a quantidade de instâncias processadas.



Podemos notar por esse gráfico que se obtém uma solução com um polinômio aproximadamente de grau (2), como podemos visualizar, a curva não cresceu muito rapidamente, comprovando desta maneira a analise de complexidade citada na seção anterior desse trabalho.

5. Conclusão

Foi possível com este trabalho aprender mais sobre paradigmas de programação como recursão e programação dinâmica, apesar da solução ser muito simples a elaboração da mesma foi um pouco custosa, achar uma maneira de representar computacionalmente este tipo de problema demandou uma quantidade considerável de tempo, pois a sua solução não é trivial. Durante a programação surgiram algumas dificuldades, como não tenho pratica com implementações de funções recursivas, tive dificuldades para fazer funcionarem corretamente.

Em relação os algoritmos programados cheguei à conclusão que a implementação por meio de funções recursivas deixa a etapa de codificação muito mais simples, acredito que a minha solução para este problema demanda muito tempo é deve existir uma maneira consideravelmente mais rápida, porem eu ao desenvolver essa solução não me deparei com outras formas para chegar à solução da mesma. A solução do algoritmo foi testada em alguns exemplos que eu mesmo desenvolvi e resolvi. Sempre o algoritmo acha a resposta certa, concluindo que o mesmo converge para a resposta em um tempo considerável.