



**Aluno:** Vinícius de Oliveira Silva

**Curso:** Ciência da Computação

**Disciplina:** Algoritmos e Estruturas de Dados III

# Documentação de Trabalho Prático

## 1. Introdução

O problema de encontrar o caminho mais curto entre dois determinados pontos é bastante comum não somente na área de Ciência da Computação, mas também no cotidiano de muitas pessoas. Partir de um determinado ponto e chegar a outro com o menor custo possível é um desafio computacional que, se resolvido de maneira eficiente, pode evitar grandes prejuízos, e por isso, tal problema vem sendo estudado há anos por diversos cientistas.

Muitas vezes, contudo, é importante conhecer não somente qual é o menor caminho entre dois pontos, mas também quantos caminhos mínimos existem entre eles.

Considerando essa abordagem, foi desenvolvido um trabalho com o objetivo de determinar quantos caminhos mínimos podem ser percorridos partindo de um ponto fixado e visando chegar a destinos arbitrários sem passar por determinados pontos previamente estabelecidos.

Para a elaboração desse trabalho, foram implementados tipos abstratos de dados (TAD's) para representar as pontos existentes no plano (que serão denominados "esquinas") e também para representar o plano propriamente dito (que será denominado "cidade"). O objetivo é calcular o número de caminhos entre uma esquina e outra sem passar por esquinas selecionadas arbitrariamente.

Espera-se com isso, praticar técnicas de programação que envolvam principalmente o conceito de programação dinâmica.

## 2. Implementação

### Estruturas de Dados:

Para a implementação desse trabalho, foram criadas algumas estruturas de dados, de forma a resolver o problema de maneira modularizada. Dessa forma, o algoritmo pode ser entendido e analisado de maneira simples. As estruturas que compõem o programa são:

- Corner;
- City;
- Queue;

O TAD Corner representa uma esquina de uma cidade, ou seja, o cruzamento entre uma rua e uma avenida (neste trabalho, entende-se por “rua”, a conexão horizontal entre as esquinas e por “avenida”, a conexão vertical entre elas).

O TAD City representa basicamente uma matriz de “Corners”, ou seja, representa a cidade a ser analisada de forma a buscar o número de caminhos mínimos entre duas esquinas.

O TAD Queue representa uma fila de esquinas e é utilizado como um auxiliar para a implementação do algoritmo de calcular o número de caminhos mínimos.

#### Representação do TAD Corner:

##### **Atributos:**

- Long Int “amntMinimalPaths”;
- Long Int “distance”;
- Short Int “isClosed”;
- Inteiro I;
- Inteiro J;

O atributo “amntMinimalPaths” é um número real que indica a quantidade de caminhos mínimos através dos quais é possível chegar àquela esquina partindo do ponto (0,0) da matriz.

O atributo “distance” é um número real que indica a distancia entre a esquina e o ponto (0,0) da matriz.

O atributo “isClosed” indica se determinada esquina está ou não bloqueada, ou seja, se é possível passar por ela ao calcular o numero de caminhos a uma esquina destino.

Os atributos “I” e “J” indicam as posições na matriz onde aquela esquina está armazenada. O valor I indica a posição horizontal e o valor J indica a posição vertical.

#### Representação do TAD City:

##### **Atributos:**

- Apontador de apontadores para Corner “cornerGrid”;
- Inteiro “streets”;
- Inteiro “avenues”;

O apontador de apontadores “cornerGrid” serve para armazenar o lugar na memória onde ficará armazenada a matriz que representa a cidade a ser analisada.

O inteiro “streets” indica quantas linhas de elementos a matriz poderá armazenar. O atributo “avenues” tem uma função parecida, servindo para indicar quantas colunas de elementos poderão ser armazenadas pela matriz.

### Representação do TAD Queue:

- Apontador para uma célula de fila “first”;
- Apontador para uma célula de fila “last”;

O apontador “first” serve para indicar a célula cabeça da fila, enquanto o apontador “last” indica qual o último elemento.

Quando a fila está vazia, ambos os apontadores apontam para a mesma posição de memória, a célula cabeça da fila.

É importante salientar que uma célula de fila é definida por uma esquina e por um apontador para a próxima célula.

### Funções e Procedimentos:

#### TAD Corner:

O TAD Corner não possui funções ou procedimentos, uma vez que essa estrutura é utilizada somente para compor outras estruturas.

#### TAD City:

**City createCity(int streets, int avenues):** Cria uma variável do tipo “City”, cuja matriz tem as mesmas dimensões dos inteiros especificados via parâmetros. A função retorna a variável criada para o seu chamador.

**void closeCorner(City\* city, int streetNumber, int avenueNumber):** A função simplesmente faz com que a esquina armazenada na matriz “city->cornerGrid”, nos índices “streetNumber”, “avenueNumber” se torne indisponível para caminhamento, ou seja, faz com que a esquina especificada seja fechada.

**void getAmntPaths (City\* city):** Essa função tem o objetivo de calcular o número de caminhos mínimos para cada uma das esquinas da cidade, assumindo que o ponto de partida é sempre a posição (0,0) da matriz. Para realizar essa tarefa, o procedimento utiliza um conceito muito interessante visto em sala, que é a busca em largura em um grafo. O algoritmo está detalhado no pseudocódigo abaixo:

```
Algoritmo CalculaNumeroCaminhos{
  Marque a esquina da posição (0,0) como já visitada;
  Armazene-a em uma fila f.
  Enquanto (f não estiver vazia), faça{
    Remova o primeiro elemento da fila e o armazene em uma variável auxiliar;
    Para cada vizinho não fechado de aux, faça{
      Se o vizinho já está na fila, faça{
        Numero de caminhos do vizinho += numero de caminhos de aux.
      }
      Se o vizinho ainda não foi visitado, faça{
        Marque o vizinho como visitado.
        Numero de caminhos do vizinho = numero de caminhos de aux.
        Insira o vizinho no final da fila f;
      }
    }
  }
}
```

**void printCity (City city):** Essa função imprime uma representação esquemática da cidade em stdout. O caractere “X” é impresso para representar esquinas fechadas, enquanto as demais são representadas por dois números indicando suas posições na matriz. É importante ressaltar que essa função não é utilizada nesse trabalho, mas sua implementação foi realizada por uma questão didática.

#### TAD Queue:

**void setQueueEmpty (Queue\* queue):** Inicializa a fila especificada por parâmetro, alocando memória para sua célula cabeça.

**int isEmpty(Queue queue):** Essa função retorna 1 ou 0 de forma a indicar se a fila especificada contém elementos além da célula cabeça.

**void insertBack(Queue \*queue, Corner corner):** Esse procedimento tem o objetivo de adicionar uma nova célula ao final da fila.

**void removeFront(Queue \*queue, Corner\* corner):** Seguindo o conceito de fila, essa função remove o primeiro elemento da fila especificada e armazena uma cópia na esquina especificada por parâmetro.

**int contains(Queue queue, Corner corner):** Essa função pesquisa a existência de um elemento especificado na fila, retornando 1 caso o encontre e 0 caso contrário.

**void printQueue(Queue queue):** Esse procedimento simplesmente imprime a fila especificada em stdout.

**void freeQueue(Queue\* queue):** Essa função libera a memória utilizada pela célula cabeça da fila. É importante que essa função seja chamada apenas quando a fila estiver vazia, pois caso contrário, todas as posições válidas da fila se tornarão inacessíveis.

#### Programa Principal:

O programa principal consiste basicamente de um loop em que, a cada iteração, resolve uma instância do problema. Em cada iteração do loop, uma configuração do problema é lida do arquivo de entrada especificado em argv[1] e resolvida através de chamadas às funções dos TAD's. O número de caminhos para cada destino desejado em cada instância é impresso no arquivo de saída especificado em argv[2] e então, toda a memória alocada para resolver o problema é liberada.

## Organização do Código, Decisões de Implementação e Detalhes Técnicos:

O código está dividido em 6 arquivos principais. Os arquivos *Queue.c* e *Queue.h* implementam o TAD Queue, já os arquivos *City.c* e *City.h* implementam o TAD City. O TAD Corner, por sua vez, é implementado somente pelo arquivo *Corner.h*.

A função *main* foi implementada no arquivo *TP2.c* e além de instanciar os TAD's, a main também é responsável pela entrada e saída de dados.

Uma importante decisão de implementação que foi tomada durante a implementação desse trabalho, foi a máxima modularização possível de cada tarefa. A implementação foi inspirada no paradigma Orientado a Objetos.

O compilador utilizado foi o MacOSX GCC, no sistema operacional Mac OSX 10.9.5 (Mavericks).

### 3. Análise de Complexidade

#### TAD Queue:

Exceto pela função "contains", todas as funções deste TAD possuem complexidade temporal e espacial  $O(1)$  pelos mesmos motivos: ausência de loops e alocações de memória constantes. Dessa forma, apenas a função "contains" será detalhada.

**Função contains:** como essa função deve percorrer toda a lista em busca de um elemento especificado, sua complexidade é  $O(n)$ .

Espaço: Como não existem alocações de memória, sua complexidade espacial é  $O(1)$ .

#### TAD City:

**Função createCity:** como a função deve percorrer uma matriz de  $n*m$  elementos, sua ordem de complexidade Temporal é  $O(n*m)$ , onde  $n$  é o número de ruas da cidade e  $m$ , o número de avenidas;

Espaço: Como deve-se alocar memória para uma matriz de  $n*m$  elementos, podemos dizer que ordem de complexidade espacial é  $O(n*m)$ .

**Função closeCorner:** como a função apenas executa comandos  $O(1)$ , e não há loops, podemos dizer que sua ordem de complexidade é  $O(1)$ .

Espaço: Como não existem declarações de variáveis ou alocações de memória no heap, a ordem de complexidade é  $O(1)$ .

**Função getAmntPaths:** esse procedimento tem o objetivo de percorrer todos os elementos da matriz e então realizar operações sobre seus vizinhos. Dessa forma, podemos dizer que no pior caso, onde não existem esquinas bloqueadas, a complexidade dessa função é  $O(n*m)$ , onde “n” e “m” representam as dimensões da matriz.

Espaço: Uma vez que apenas variáveis estáticas são declaradas e não há alocações de memória, podemos dizer que a ordem de complexidade espacial dessa função é  $O(1)$ .

**Função freeCity:** como a função executa um loop n vezes, onde n é o número de ruas da cidade, podemos dizer que sua complexidade é  $O(n)$ .

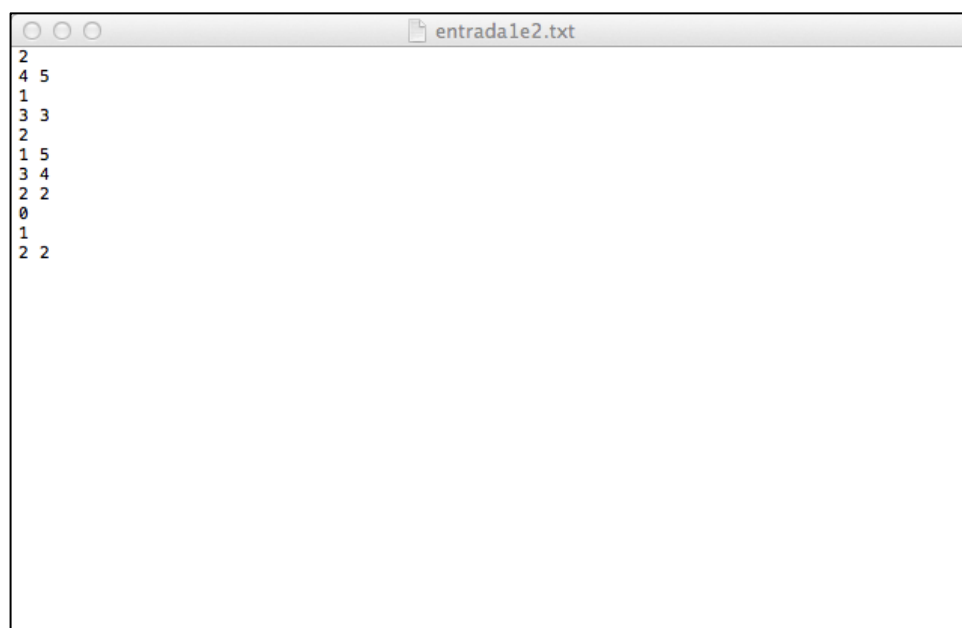
Espaço: Como essa função apenas libera memória, podemos dizer que sua complexidade espacial é  $O(1)$ .

## 4. Testes

Diversos testes foram realizados com o programa no intuito de avaliar seu comportamento em diversas configurações de cidade, tais como destinos inacessíveis, destinos que requerem que se percorram caminhos em todas as direções, etc.

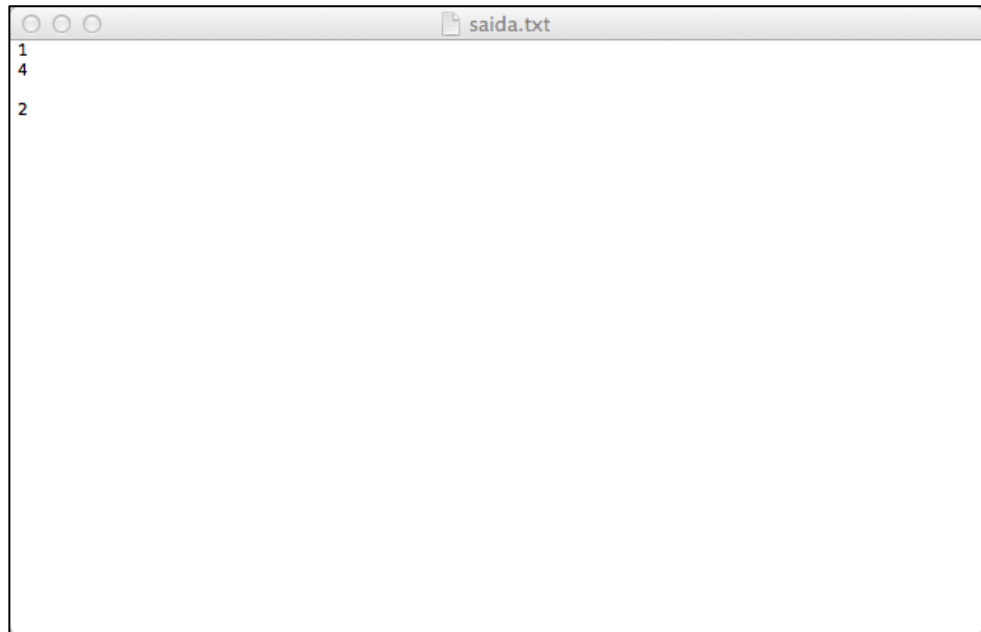
Os testes foram executados em um computador equipado com processador Intel Core 2 Duo @2.4GHz e 4GB de memória. Um exemplo de teste realizado está mostrado abaixo:

Entrada utilizada:



```
2
4 5
1
3 3
2
1 5
3 4
2 2
0
1
2 2
```

Saída gerada:



## 5. Conclusão

A implementação desse trabalho não ocorreu tão bem como o esperado, uma vez que, devido à inexperiência, foi bastante difícil identificar que o problema teria solução relativamente simples através do uso de busca em largura em grafos. Depois que tivemos uma aula sobre esse assunto, contudo, a solução do trabalho fluiu de maneira mais tranquila, uma vez que o problema foi melhor compreendido.

Podemos dizer que o trabalho cumpriu bem os seus objetivos, que incluíam a prática de técnicas de programação dinâmica e o desenvolvimento do raciocínio lógico.

Obs.: Apesar do trabalho ter sido compilado usando o comando UNIX make, e assim, o uso de IDE's ser dispensável, é importante salientar que o desenvolvimento do código fonte foi feito no ambiente eclipse, uma vez que tal IDE oferece inúmeras facilidades na edição de códigos-fonte. O makefile utilizado, no entanto, foi criado a partir do ambiente gedit, e não gerado automaticamente pelo IDE.

## 6. Referências

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