Grafo – algoritmo de menor distância

Estrutura de Dados – turma E - Professor: George Teodoro

Autores:

Bruno Fernandes Carvalho – 150007159

Leonardo Nunes Cornelio Rêgo- 150015046

Introdução

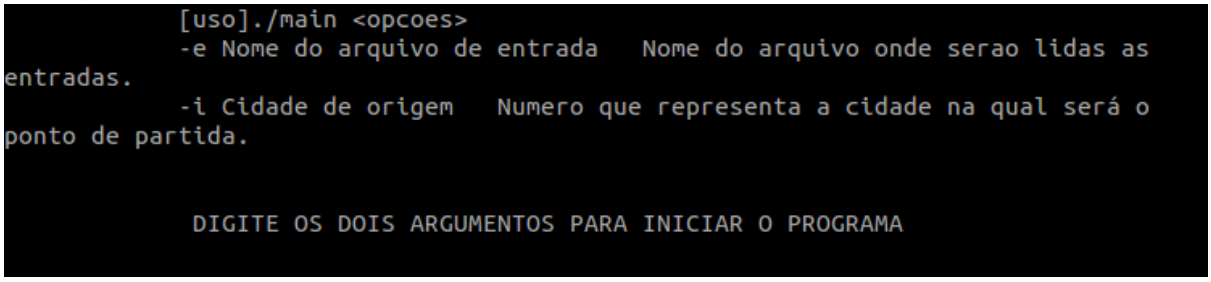
A finalidade desse trabalho é representar um grafo de forma abstrata em alguma estrutura de dado e calcular a menor distância total ligando todos os vértices do grafo a partir de um vértice determinado previamente. Nesse problema, cada vértice representa uma cidade e cada aresta é o caminho com determinado custo ligando duas cidades, ou seja, uma aresta não direcionada. No final, será mostrado ao usuário o menor caminho que deve ser percorrido no grafo sem que passe duas vezes pela mesma cidade, assim como o peso total mínimo para alcançar todas essas cidades a partir da origem.

Como executar o programa

Para executar o programa, foi usado o utilitário Make, já instalado no sistema operacional Linux. Como foram utilizados quatro módulos, foi necessário criar quatro objetos (arquivo .o) para relacioná-los e criar um executável chamado “main”. A seguir, mostra-se como deve ser gerado o arquivo makefile para a correta compilação dos módulos.

```
main: main.o logica.o in_out.o estruturas.o
	gcc logica.o main.o in_out.o estruturas.o -o main
main.o: main.c logica.h estruturas.h
	gcc -g -c main.c
logica.o: logica.c logica.h
	gcc -g -c logica.c
in_out.o: in_out.c in_out.h
	gcc -g -c in_out.c
estruturas.o: estruturas.c estruturas.h
	gcc -g -c estruturas.c
```

Como foi usado no programa a função getopt para receber argumentos da linha de comando, é necessário definir todos eles na hora de executar a “main”. A seguir, é mostrado o menu de ajuda para executar o programa.



Assim, na linha de comando, deve ser escrito para executar:

./main -e entrada.txt -i 2

2 é a cidade de origem e entrada.txt é o arquivo de entrada contendo as informações para a formação do grafo. O arquivo deve conter o número de cidades na primeira linha do arquivo. Nas demais, deve ser representado as duas cidades ligadas pela aresta, assim como o custo dessa distância, sendo cada um desses três dados numa linha do arquivo, conforme mostrado a seguir:

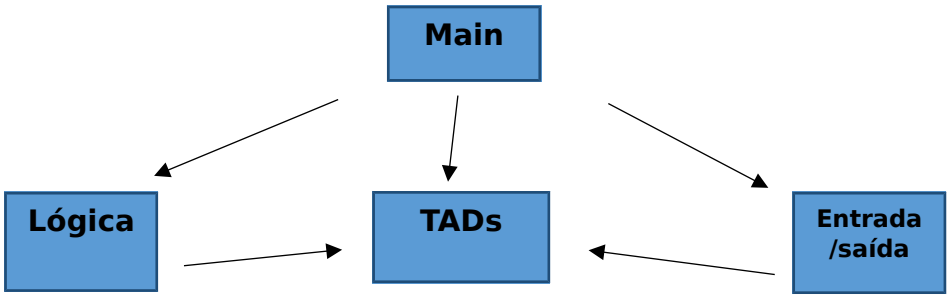
```
4
0 1 4
0 2 3
1 3 2
1 2 1
```

No exemplo acima, na primeira linha, 4 representa a quantidade de cidades no grafo, e na linha seguinte, mostra-se que as cidades 0 e 1 são ligadas com uma aresta de peso 4. Além disso, para o algoritmo resolver de forma correta o problema inicial, deve ser gerado no arquivo de entrada grafos conexos, ou seja, de forma que a partir de qualquer vértice seja possível alcançar todos os outros. Também deve ser escolhido uma cidade de origem coerente com o grafo proposto.

Se os grafos não forem conexos ou não existir um caminho que ligue todas as cidades sem retornar a cidades já percorridas, a seguinte mensagem será fornecida : “Não existe caminho que conecte todas as cidades sem passar 2 vezes em uma mesma cidade”. Além disso, se existirem mais de um caminho mínimo com custos iguais, o algoritmo irá ignorar e representará apenas um deles.

Descrição dos módulos

O algoritmo foi dividido em quatro módulos: um que contém o programa principal (main), um que contém a lógica do algoritmo de ordenação, um que gerencia a entrada e saída de dados e o último que contém os TADs (tipo abstrato de dados) e as estruturas de dados utilizadas. O programa principal depende de todos os módulos, além destes serem interdependentes, como mostrado na figura a seguir:



-Programa principal (main)

A main é responsável por articular as chamadas da maioria das funções para o funcionamento correto do código. Nela, também é usado a primitiva getopt para receber argumentos da linha de comando do terminal. Essa função recebe o nome do arquivo texto de entrada e a cidade de origem. Esse módulo também é responsável por declarar as estruturas de dados usadas, chamar as funções que gerenciam a entrada e saída de dados, assim como as funções que contém a lógica do programa.

-Lógica do programa

Essa parte do programa contém as funções responsáveis por terem toda a lógica do algoritmo. Entre elas, a mais importante é a “dfs_visit”, já que nela está implementada a lógica principal para solucionar o problema. Foi utilizado “Recursive Backtracking”, ou seja, um método recursivo eficiente para resolução de problemas, em que testa se um caminho é a solução; se não for, o algoritmo volta até certo ponto do grafo para testar novas soluções. O cabeçalho de todas funções é mostrado a seguir:

```
int verifica_vertices(tipolista vetor[], int n);
void montar_lista(tipolista *lista_aux,tipolista vetor[], int u);
void dfs_visit(tipolista *lista_aux,tipolista vetor[], int u, int n);
```

-Entrada e saída de dados

Esse módulo é responsável somente por organizar a entrada e saída de dados, além de mostrar o menu de ajuda relacionado com a função getopt. O cabeçalho das funções é mostrado abaixo:

```
void mostra_ajuda(char *name);
```

```
void saida(tipolista *lista_aux);
void le_cidades(FILE **fp, int *qnt);
int le_arestas(FILE **fp,int *i, int *x, int* p);
```

-TADs e estrutura de dados

Nessa parte do algoritmo estão os tipos abstratos de dados (especificação matemática que determina a implementação das estruturas de dados) e as próprias estruturas. Dois tipos são criados: tipolista que define a lista e tipocelula contendo os elementos da célula e um apontador para a célula seguinte da lista. Também foi criado um tipo apontador, que aponta para as células criadas. Essas estruturas são mostradas a seguir:

```
typedef struct tipocelula *apontador;


typedef struct tipocelula{
   int origem,destino, peso;
   apontador prox;
 }celula;


 typedef struct{
   char color;
   int pai;
   int soma;
   apontador primeiro, ultimo;
 }tipolista;
```

Além disso, diversas operações matemáticas são feitas com as estruturas declaradas no programa principal. Entre elas, pode-se citar a inserção de uma aresta nova na lista adjacente de determinado vértice, a inserção das células na lista auxiliar, a liberação de memória das listas de ponteiros criadas e funções responsáveis por criar listas e vetores vazios. O cabeçalho de cada uma dessas funções é mostrado abaixo:

```
void insere(tipolista vetor[], int i, int x, int p);
```

```
void insere_aux(tipolista *lista_aux, tipolista vetor[], int u, int flag);
void libera(tipolista vetor[], int n);
void libera_aux(tipolista *lista_aux);
void flvazia(tipolista *lista);
void fvvazio(tipolista vetor[],int i);
```

Análise dos algoritmos e complexidade

Cada módulo possui rotinas que apresentam papel fundamental na execução correta do programa, e serão analisados os algoritmos com o objetivo de entender as escolhas feitas, entender como eles funcionam e seus pontos fortes e fracos. Além disso, será analisado a complexidade dessas funções para determinar a otimização do código.

-Programa principal (main)

O programa principal é simples e tem responsabilidade de chamar as funções apropriadas para leitura de dados e para imprimí-los. Além disso, recebe-se os parâmetros digitados na linha de comando pelo usuário usando Getopt.

Na coordenação da lógica do programa, ela organiza a entrada de dados do arquivo num vetor de listas adjacentes (representação do grafo), utilizando a função “Insere” do módulo de estruturas de dados. Depois, chama a função “dfs_visit”, responsável por resolver o problema. Quando o programa retorna à “main”, ela só precisa liberar a memória das listas criadas chamando métodos do módulo “Tads”.

Análise de complexidade

A maioria das manipulações da “main” são simples e apresentam ordem de complexidade constantes. Porém, durante a leitura dos dados de entrada, há um loop que tem O(E), já ele itera enquanto tiver arestas a serem adicionadas no grafo. Além dessa, tem um trecho que itera V vezes, sendo V a quantidade de vértices. Nessa parte, o vetor que representa o grafo é inicializado vazio e deve percorrer todos os vértices, representando O(V).

A parte mais custosa desse módulo é justamente a que está atrelada à lógica principal do programa: a função recursiva “dfs_visit”. Ela possui ordem de complexidade para o caso médio de O(V + E), sendo E a quantidade de aresta e V a quantidade de vértices.

-Lógica do programa

A lógica implementada para resolver o desafio de percorrer o menor caminho passando por todas as cidades foi mesclar o código de Busca em Profundidade em um grafo e Recursive Backtracking. Nesse método, busca-se um caminho possível que percorra todos os vértices; se isso acontecer, esse caminho provisório é armazenado em uma lista auxiliar, que contém as arestas que formam esse percurso. Se todos os vértices não forem alcançados, essa opção de caminho é descartada, e outros são procurados, usando recursividade. A implementação será detalhada a seguir.

A função “dfs_visit” procura por profundidade os vértices adjacentes, usando o mesmo algoritmo “DFS” já conhecido na literatura. Quando não há mais opções, ou seja, todos os vértices adjacentes estão cinzas, verifica-se com a função “verifica_vertices” se todos as cidades estão incluídas no caminho, e se estiverem, a função auxiliar “montar_lista” é encarregada de criar a lista auxiliar contendo as arestas desse caminho. Porém, não há garantias que esse é o caminho com menor custo, então foi necessário alterar o código “DFS”: em vez do vértice ficar cinza, indicando que já foi percorrido, ele volta a ficar branco, pois durante o Recursive Backtracking ele pode ser alcançado por outro caminho, e deve estar branco para que a busca por profundidade dê certo e chegue até ele.

Assim, se um novo caminho existir, compara-se o peso total do novo com o armazenado na lista auxiliar. Se for menor, libera-se a memória da lista antiga e cria-se uma nova, contendo as novas arestas que formam o melhor caminho. Desse jeito, garante-se que será alcançado o menor caminho passando por todos os vértices de maneira eficiente.

Análise de complexidade

Conforme visto na análise de complexidade do programa principal, a função “dfs_visit” tem O(V+E) para o caso médio, já que na média passa-se por todos os vértices e todas as aresta do grafo para verificar o melhor caminho. Vale lembrar que as constantes são desconsideradas na análise de complexidade das funções.

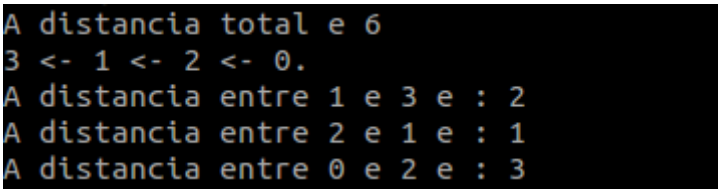
Ainda, a função “verifica_vertices” possui O(V) no pior caso, pois deve percorrer o vetor que abstrai o grafo no máximo V vezes, verificando se todos os vértices do grafo foram alcançados e se representam um caminho adequado para o problema. Além dessas, a função que monta a lista auxiliar apresenta O(1) para o melhor caso, que acontece quando está sendo criado pela primeira vez uma lista auxiliar, e O(V) no pior caso, que acontece quando já existe outro caminho na lista auxiliar. É necessário percorrer as arestas adjacentes dos V vértices que formam o

percurso da lista auxiliar já existente, para fazer a comparação da soma dos pesos.

-Entrada e saída de dados

O algoritmo de entrada e saída de dados é bastante simples, contendo apenas funções responsáveis por receber os dados de entrada, imprimir os dados de saída e o menu de ajuda para executar corretamente o programa pelo terminal.

Há duas funções responsáveis por ler o arquivo de entrada: uma que lê a primeira linha contendo a quantidade de cidades e outra que lê as informações das arestas não direcionadas que ligam as cidades. Já na parte de saída de dados, a função “saida” recebe a lista auxiliar que contém o menor caminho passando por todos os vértices, conforme descrito no módulo da lógica do algoritmo. Nela, mostra-se o custo total do caminho a partir da origem definida, a ordem das cidades a serem percorridas e as distâncias parciais entre cada cidade, conforme mostrado abaixo:



Nesse exemplo, a origem é a cidade 0 e as setas indicam o sentido do percurso. Na primeira linha é visto a distância total mínima e nas demais linhas as distâncias parciais.

Análise de complexidade

Como a maioria das funções desse módulo não apresentam iteração e são apenas responsáveis por ler ou mostrar dados para o usuário, estas apresentam O(1), indicando que a ordem de complexidade é constante e não é um custo alto no tempo de execução do programa. Apesar disso, alguns trechos da função que mostra os dados para o usuário apresentam “loops” e possuem O(V), já que passam por uma aresta adjacente de cada vértice na hora de imprimir o caminho de menor custo para o usuário.

-TADs e estrutura de dados

O algoritmo descrito nesse módulo é responsável por dar suporte à lógica do programa. Aqui são definidas diversas funções que fazem operações com os TADs, além do próprio tipo e das estruturas que estão contidas nele.

A estrutura escolhida para representar o grafo foi uma lista adjacente. Assim, foi criado um vetor que contém em cada posição um vértice do grafo e uma lista adjacente contendo todas as arestas ligadas a ele, e consequentemente, os vértices adjacentes. Quando o usuário determina uma aresta entre dois vértices, essa aresta é adicionada na lista adjacente de ambos os vértices, já que corresponde à um grafo não direcionado. Assim, pode-se concluir que as células das listas adjacentes correspondem às arestas, e seus elementos “origem”, “destino” e “peso” são características dela. “Origem” representa o vértice de origem, “destino” o vértice final e “peso” o custo da aresta. Como escolheu-se usar essa estruturação, foi necessário utilizar mais memória para a representação do grafo, já que cada aresta é adicionada à lista de dois vértices. Isso pode ser desotimizado em relação ao custo de memória utilizado para solucionar o problema proposto.

Além disso, cada vértice possui determinadas propriedades: “color” é uma variável que auxilia a execução do algoritmo e “pai” representa o pai do vértice e é essencial na ligação das cidades para formar o caminho. “Soma”, que é uma variável que dá suporte à lista auxiliar descrita na lógica, representa a soma dos pesos do caminho feito.

As demais funções responsáveis por operar e alterar as estruturas de dados também são importantes. “Insere” é responsável por adicionar uma aresta nova na lista adjacente de um vértice, “Insere_aux” insere as arestas na lista auxiliar que contém o menor caminho provisório, “libera” e “libera_aux”, que liberam, respectivamente, a memória alocada dinamicamente das listas adjacentes de cada vértice e da lista auxiliar, e as outras duas funções que restaram (“fvvazio” e “flvazia”), em que a primeira cria um vetor vazio, inicializado os vértices, e a segunda, que cria uma lista de apontadores vazia.

Análise de complexidade

A complexidade da maioria dessas funções são simples e apresentam O(1), já que elas consistem em testes de condição e atribuições, que são operações que não têm custo alto para o tempo de execução.

Porém, a função que insere as arestas na lista auxiliar apresenta O(V), já que são inseridas nessa lista uma aresta adjacente de cada vértice, representando um possível caminho que passa por todas cidades. Além dessa, outra função que tem iteração é a que libera os vértices da lista auxiliar, e também é O(V).

Além dessas, a mais custosa do módulo é a que libera as listas adjacentes de cada vértice. Como já dito anteriormente, um ponto fraco da representação do grafo é a atribuição de uma mesma aresta à lista dos dois vértice que ela conecta. Assim, na hora de desalocar essas lista, é necessário passar por todos os vértices e cada uma dessas listas, representando uma ordem de complexidade média de O(V + E). Mesmo que cada aresta seja alocada em dobro na memória, esse aumento é uma constante e é desconsiderado na análise da complexidade.

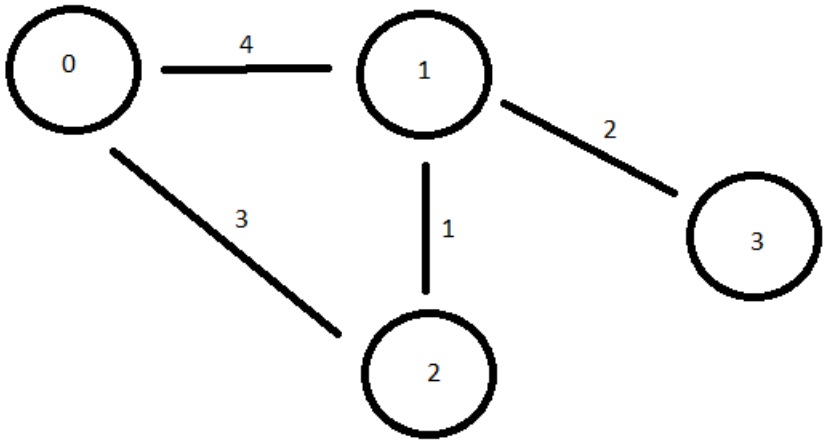
Análise de dados

Iremos agora analisar alguns testes para confirmar a corretude do programa.

1. Teste 1

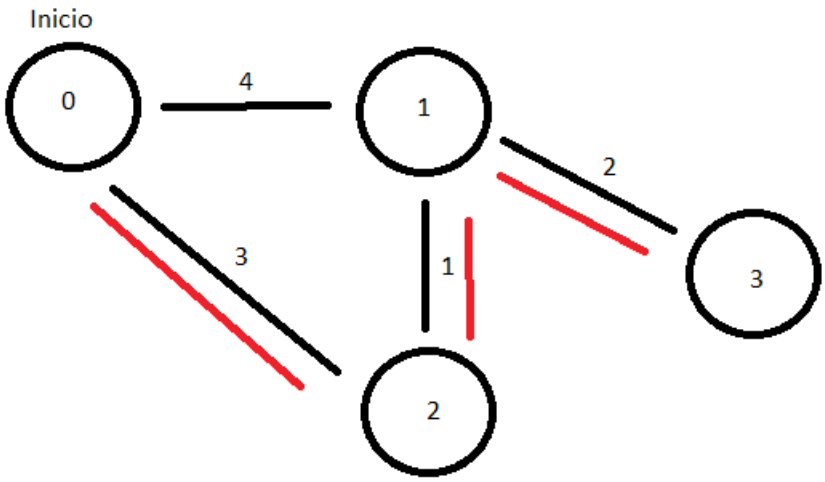
```
4
1 2 1
1 3 2
0 2 3
0 1 4
```

Montou-se o grafo a seguir para mostrar visualmente o real significado das entradas e verificar se o caminho encontrado pelo algoritmo é realmente o mais curto possível.



Ao se executar o programa, o resultado encontrado, tendo como base o vértice 0 é: 0 → 2 → 1 → 3 com distância total de 6.

Analisando esse grafo, nota-se que partindo do vértice 0 só existe um caminho possível para percorrer todos os vértices passando apenas uma vez em cada.

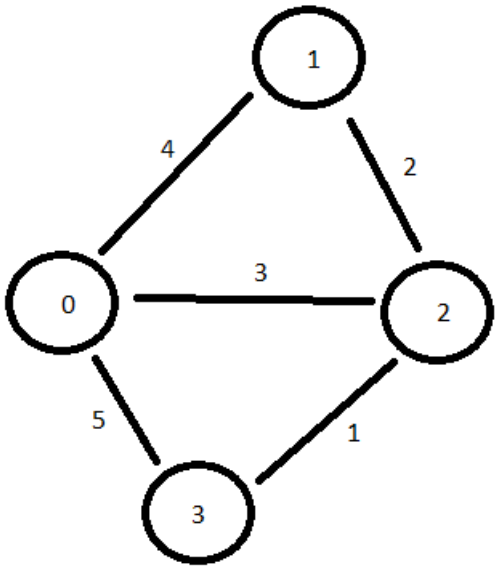


Logo, o programa conseguiu encontrar o caminho correto para o teste 1.

2. Teste 2

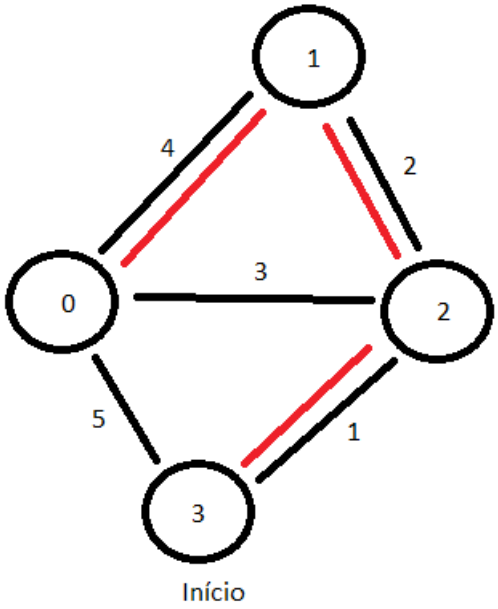
```
4
3 0 5
3 2 1
2 1 2
2 0 3
0 1 4
```

Montou-se o grafo a seguir para mostrar visualmente o real significado das entradas e verificar se o caminho encontrado pelo algoritmo é realmente o mais curto possível.



Ao se executar o programa, o resultado encontrado, tendo como base o vértice 3 é: 3 → 2 → 1 → 0 com distância total de 7.

Analisando esse grafo, nota-se que partindo do vértice 3 existem 4 caminhos possíveis, porém apenas 1 deles nos interessa, ou seja, o mais curto. Vale ressaltar que se existissem 2 caminhos com a mesma distância total, o algoritmo selecionaria apenas uma, aleatoriamente.

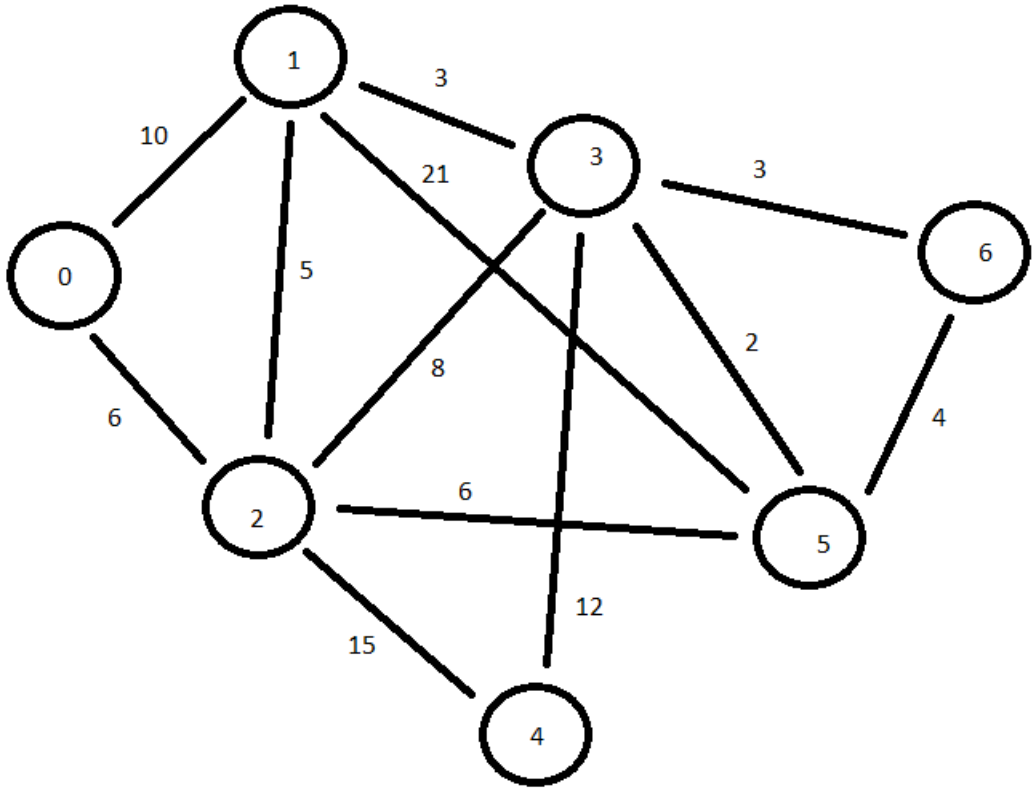


Logo, o programa conseguiu encontrar o caminho correto para o teste 2.

3. Teste 3

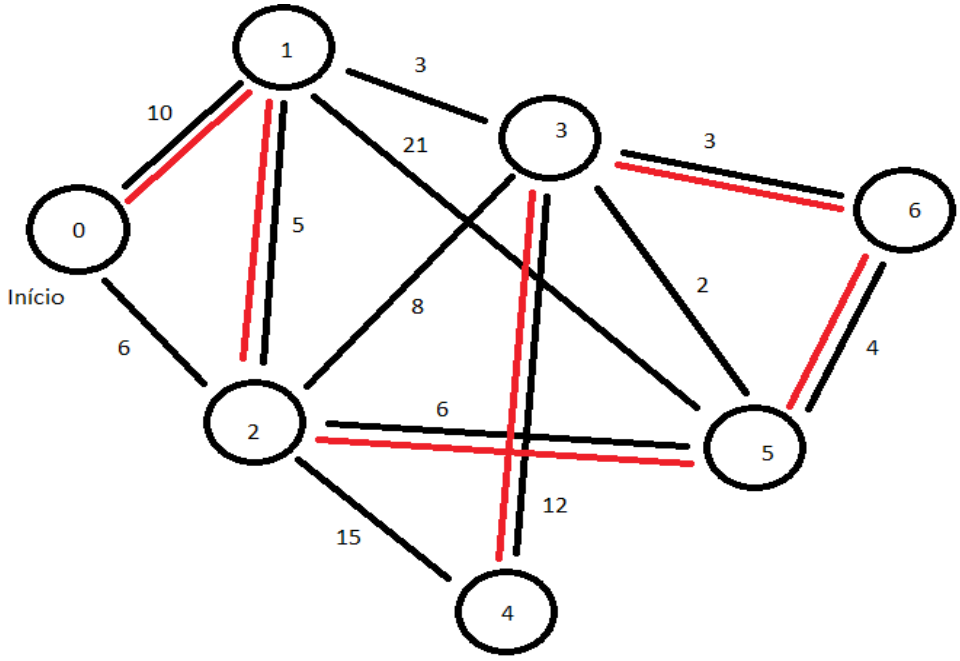
```
7
0 1 10
0 2 6
1 3 3
1 2 5
1 5 21
2 3 8
2 4 15
2 5 6
3 6 3
3 5 2
3 4 12
5 6 4
```

Montou-se o grafo a seguir para mostrar visualmente o real significado das entradas e verificar se o caminho encontrado pelo algoritmo é realmente o mais curto possível.



Ao se executar o programa, o resultado encontrado, tendo como base o vértice 0 é: 0 → 1 → 2 → 5 → 6 → 3 → 4 com distância total de 40.

Analisando esse grafo, nota-se que partindo do vértice 0 existem múltiplos caminhos possíveis, porém apenas 1 deles nos interessa, ou seja, o mais curto.

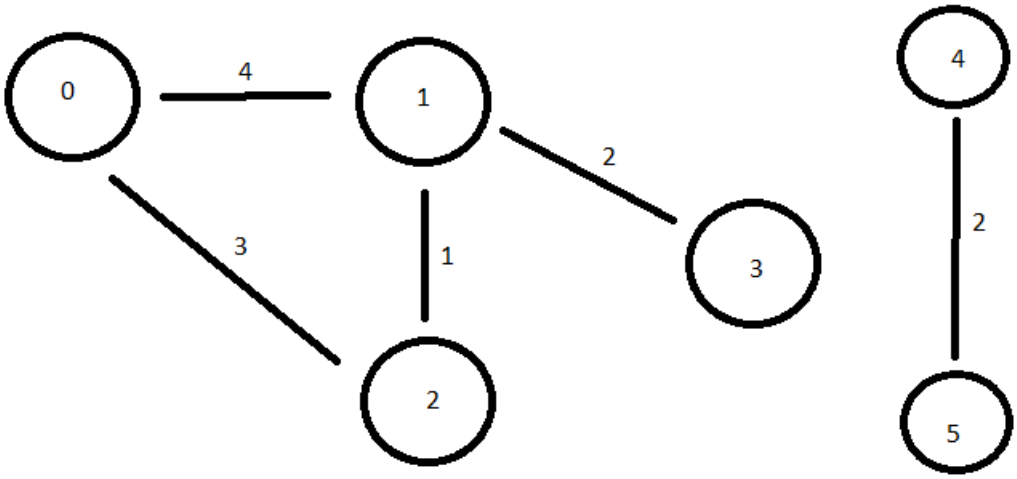


Logo, o programa conseguiu encontrar o caminho correto para o teste 3.

4. Teste 4

```
6
1 2 1
1 3 2
0 2 3
0 1 4
4 5 2
```

Montou-se o grafo a seguir para mostrar visualmente o real significado das entradas e verificar se o caminho encontrado pelo algoritmo é realmente o mais curto possível.



Ao se executar o programa, o resultado encontrado, tendo como base qualquer vértice, será caminho inexistente, pois o grafo apresentado não é conexo, ou seja, é impossível acessar todos os vértices a partir de um vértice qualquer.

Conclusão

Conforme visto, o algoritmo resolve de forma eficiente o problema de menor distância passando por todas as cidades usando Recursive

Backtracking. Ficou claro que mesmo que o primeiro caminho alcançado não seja o melhor, o algoritmo consegue avaliar isso e buscar por outras soluções. Pela análise de dados, sabe-se também da corretude do código, demonstrando que os resultados são os esperados e o problema é resolvido de forma satisfatória.