Engenharia e Ciência da Computação – Estruturas de Dados

Trabalho 1 – Caminho Mínimo entre Cidades

Informações Gerais

- **Data Limite de Entrega**: 27/11/2023 (23:59).
- **Pontuação**: 10 pontos (20% da nota do semestre).
- **Formato de Entrega**: Os arquivos produzidos no trabalho devem ser compactados em formato **.zip** e submetidos na tarefa do ambiente virtual de aprendizagem (AVA).
- Os trabalhos devem ser desenvolvidos individualmente.
- **Importante**: Trabalhos entregues após a data limite sem justificativa com comprovação documental (atestado médico, etc.), ou que não estiverem de acordo com a especificado receberão nota zero.

Contextualização

Suponha que você foi contratado para atuar como desenvolvedor em uma empresa da área de logística de transporte. A maioria dos clientes da empresa são fábricas que precisam distribuir seus produtos para os pontos de venda. Os pontos de venda estão localizados em diferentes bairros. Com objetivo de minimizar o tempo de entrega e o consumo de combustível, você foi incumbido de escrever um algoritmo para encontrar o menor caminho da fábrica até cada um dos pontos de entrega.

A Figura 1 ilustra as distâncias entre a fábrica e os pontos de entrega. Estas relações são representadas por um grafo em que o nó verde representa a fábrica, os nós azuis representam os pontos de entrega e as arestas indicam que existe um caminho entre os nós e o tamanho (custo) do caminho é o valor associado à aresta.

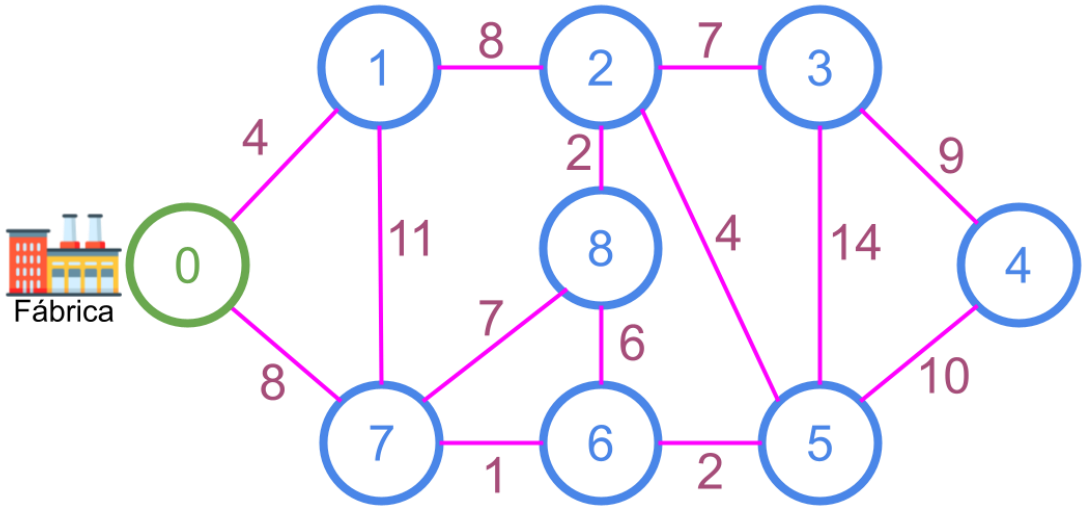

Figura 1: Grafo representando as distâncias entre a fábrica e os pontos de entrega.

Para alcançar um ponto de entrega pode ser necessário passar por outros bairros. Por exemplo, não é possível ir diretamente de 0 para 2. Mas é possível alcançar o ponto de entrega 2 via os caminhos:

- 0 -> 1 -> 2,
- 0 -> 7 -> 8 -> 2,
- 0 -> 7 -> 6 -> 8 -> 2,
- etc.

O objetivo do seu programa é encontrar os caminhos mais curtos saindo do nó 0 para todos os outros nós do grafo.

Entradas e Saídas

A entrada do programa será um arquivo texto com o formato abaixo:

Entrada
8 1 4 7 8 0 4 2 8 7 11 1 8 8 2 3 7 5 4 2 7 5 14 4 9 3 9 5 10 2 4 3 14 4 10 6 2 5 2 7 1 8 6 0 8 1 11 6 1 8 7 2 2 6 6 7 7

A primeira linha contém um número N indicando quantos de nós existem do grafo. Em seguida, cada linha representa as conexões existentes para um nó. A primeira das N linhas são as conexões do nó 0, a segunda são as conexões do nó 1 e assim por diante. Para cada conexão, são dados dois valores: um inteiro representando o nó de destino e um *float* representando o custo do caminho entre os nós. Assumiremos que a fábrica é sempre o nó 0 e que todos os demais nós contêm pontos de entrega.

O programa deve gerar como saída na tela N-1 linhas (N-1 porque a fábrica não é um destino), cada uma contendo a sequência de nós (o caminho) de menor custo entre a fábrica e o ponto de entrega, seguido do custo total do caminho com duas casas após a vírgula. A primeira linha deve ser o caminho de custo mínimo da fábrica para o nó 1, a segunda linha deve ser o caminho de custo mínimo da fábrica até o nó 2 e assim por diante.

A saída esperada para a entrada apresentada acima é:

Saída
0 -> 1: 4.00 0 -> 1 -> 2: 12.00 0 -> 1 -> 2 -> 3: 19.00 0 -> 7 -> 6 -> 5 -> 4: 21.00 0 -> 7 -> 6 -> 5: 11.00 0 -> 7 -> 6: 9.00 0 -> 7: 8.00 0 -> 1 -> 2 -> 8: 14.00

Desenvolvimento

O algoritmo de Djikstra deve ser usado para encontrar os caminhos mínimos entre a fábrica e os pontos de entrega. Ele opera em um loop principal em que a cada iteração um nó é selecionado para ser visitado. O nó escolhido é aquele com menor distância para a origem dentre os que não visitados antes. Para cada vizinho não visitado do nó, é calculada a distância da origem para o vizinho passando pelo nó e, se esta distância for menor do que a dos caminhos alternativos para o vizinho encontrados até o momento, ela é registrada no vizinho como sendo o menor caminho até então. Em iterações posteriores, caminhos melhores para o vizinho podem ser encontrados. Quando um nó é selecionado para ser visitado, necessariamente não existem mais caminhos alternativos que podem ser melhores. Portanto, o melhor caminho para o nó já pode ser retornado. A Tabela 1 traz o pseudocódigo do algoritmo de Djikstra.

Tabela 1: Pseudocódigo do Algoritmo de Dijkstra

Algoritmo Dijkstra(Grafo, Origem):

1. Crie um vector de valores booleanos chamado "Visitados" com tamanho igual ao número de nós. Inicialize os valores como falso e quando um nó for visitado, mude o valor para 1.
2. Crie uma fila de prioridade "NaoVisitados" com o nó origem e distância para a origem zero.
3. Crie um vector do tamanho do número de nós para armazenar o antecessor de cada no seu caminho mínimo (de onde ele veio) e o custo para o antecessor.
4. Enquanto houver vértices não visitados e a fila de prioridades não for vazia:
5. Escolha o nó não visitado U com a menor distância para a origem em "NaoVisitados" e o marque como visitado (adicione-o no conjunto "Visitados"). Atualize também o vetor de antecessores para adicionar o antecessor de U no caminho que foi retirado do *heap* e o custo de U para o antecessor.
6. Para cada vizinho não visitado V de U:
7. Calcule a distância de V para a origem passando por U somando a distância de U para a origem e a distância entre U e V.
8. Adicione o par (V, distância) em "NaoVisitados".
9. Retorne a estrutura com os caminhos mínimos.

Ele começa criando um *array* booleano chamado "Visitados," que possui um elemento para cada nó no grafo. Inicialmente, todos os valores são definidos como falso para indicar que nenhum nó foi visitado. Conforme o algoritmo progride, os nós visitados são marcados como verdadeiros no *array.*

Em seguida, é criada uma fila de prioridade chamada "NaoVisitados" que conterá os nós a serem explorados. A fila de prioridade é usada para a seleção eficiente do próximo nó a ser explorado. **Um *heap* deverá ser utilizado para implementar a fila de prioridade**. O *heap* organiza os nós não visitados de acordo com suas distâncias atuais em relação à origem, de modo que o nó com a menor distância esteja sempre no topo da estrutura. Isso permite ao algoritmo escolher o próximo nó a ser explorado de forma eficiente, sem ter que verificar todas as opções. As operações de inserção e extração do nó a ser visitado do *heap* não devem ter complexidade superior a O(log N).

Poderá deverá ser criada uma estrutura de dados para armazenar os antecessores de cada nó e os respectivos custos no melhor caminho da origem até o nó. Esta estrutura permite recuperar o melhor caminho partindo de todos iterando pelos antecessores, começando do nó e indo até a origem, sem a necessidade de armazenar cópias de trechos de caminhos.

O loop principal do algoritmo consiste nos passos descritos a seguir. O nó de origem é adicionado ao *heap* (fila de prioridade) com uma distância para a origem de zero. Enquanto houver vértices não visitados no grafo e a fila de prioridades "NaoVisitados" não estiver vazia, o nó U com a menor distância à origem e que ainda não foi visitado é selecionado de "NaoVisitados." O nó U é, então, marcado como visitado e o menor caminho até ele é registrado. A seguir, o algoritmo itera sobre todos os vizinhos não visitados de U. Para cada vizinho V, o algoritmo calcula a distância de V à origem passando por U. Isso é feito somando a distância de U à origem com a distância entre U e V. O nó V é adicionado no *heap* usando a distância como prioridade. Note que um determinado nó pode ser inserido várias vezes no *heap*. Contudo, pela propriedade do *heap*, o caminho de menor custo para cada nó terá prioridade sobre os caminhos alternativos. O processo de visitar os nós com menor custo para a origem, marcá-los como visitados e calcular as distâncias para os vizinhos e adicionar os vizinhos no *heap* se repete até que todos os nós tenham sido visitados ou não haja mais nós em "NaoVisitados." Finalmente, o algoritmo retorna a estrutura de dados que contém os antecessores dos nós nos melhores caminhos e esta estrutura de dados é utilizada para exibir a saída na tela.

Regras

- As estruturas de dados utilizadas para armazenar coleções de itens (vector, listas encadeadas, heaps, etc.) devem ser opacas e genéricas. Demais estruturas de dados específicas do problema **não** precisam ser nem opacas nem genéricas.
- Busque modularizar o código e nomear variáveis e funções de forma a torná-lo simples e legível. Estruturas de dados além daquelas utilizadas para armazenar coleções de itens podem contribuir neste objetivo. A título de exemplo, a Tabela 2 ilustra um arquivo main.c com boa modularização.
    - As estruturas Problem e Path foram utilizadas para armazenar dados do problema e os caminhos resultantes do algoritmo de Dijkstra, respectivamente.
    - O arquivo dijkstra.h expõe apenas a função djikstra_solve que recebe como entrada um Problem e retorna um Vector de Paths. Detalhes de implementação e funções auxiliares do algoritmo estão ocultos no arquivo dijkstra.c .
    - Note como a boa escolha de nomes de funções torna fácil compreender a sequência de passos realizados na função main.c.
- Busque ser consistente no estilo de escrita do código. Um exemplo conciso de regras de estilo é dado a seguir. Você não precisa seguir estas regras específicas, mas é importante que você siga algum estilo específico.
    - Funções, variáveis e atributos de estruturas devem ser escritos em minúsculas e usando *underline* (“_”) para separação de palavras.
    - Nomes de estruturas devem utilizar UpperCamelCase (ou PascalCase). Isto é, as palavras que compõe o nome devem iniciar com maiúsculas e devem ser unidas sem espaços.
    - Blocos de código em uma função devem ser separados por uma linha vazia. Um exemplo seria separar a declaração de variáveis do primeiro *loop* ou condicional da função. Funções devem ter menos de uma tela de tamanho.

- Duas linhas vazias devem ser utilizadas para separar funções e para separar os includes das instruções iniciais em um arquivo.
- Comentários devem ser utilizados para explicar trechos de código obscuros ou para documentar a interface (entradas e saídas) de funções. Demais trechos de código devem ser escritos de forma a serem compreensíveis sem a necessidade de comentários.

Tabela 2: Exemplo de arquivo main.c em que foi realizada uma boa modularização usando estruturas de dados e funções.

```c
#include <stdio.h>
#include <stdlib.h>
#include "heap.h"
#include "vector.h"
#include "problem.h"
#include "djikstra.h"
#include "path.h"


void print_and_destroy_paths(Vector *paths)
{
	for (int i = 0; i < vector_size(paths); i++)
	{
		Path *path = vector_get(paths, i);
		path_print(path);
		path_destroy(path);
	}


	vector_destroy(paths);
}


int main()
{
	Problem *problem_data;
	Vector *paths;


	problem_data = problem_data_read("in.txt");
	paths = djikstra_solve(problem_data);
	print_and_destroy_paths(paths);
	problem_data_destroy(problem_data);


	return 0;
}
```