Trabalho Prático 1
Algoritmos I - Aritmofobia

Wallace Eduardo Pereira - 2021032013

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

wallaceep@ufmg.br

1. Introdução

O problema proposto foi implementar um algoritmo que simule um mapa por meio de um grafo, onde os vértices representam as cidades e as arestas representam um caminho entre essas cidades, sendo essas arestas ponderadas, em que o peso de uma aresta representa a distância entre duas cidades. Dado o grafo, o objetivo do problema é encontrar a menor distância entre a primeira e a última cidades, passando somente por distâncias pares, e além disso, passando por um número par de caminhos.

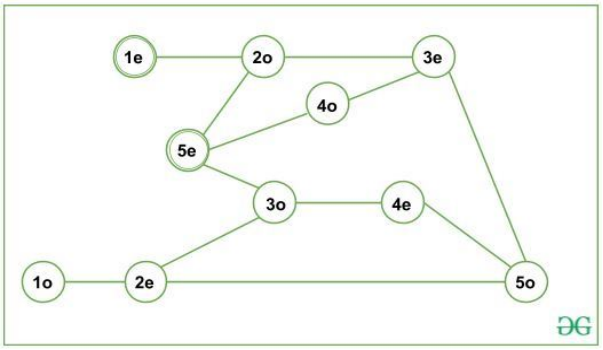
2. Implementação

O programa foi desenvolvido na linguagem C++ e compilado pelo compilador G++ da GNU Compiler Collection.

3. Modelagem

Para resolver o problema proposto, primeiro foi armazenado o grafo em uma lista de adjacência de tamanho n (sendo n a quantidade de cidades, ou seja, o número de vértices), e depois foi criado um grafo bipartido de tamanho 2n. Em seguida, foi utilizado o algoritmo Dijkstra, que encontra o caminho de custo mínimo entre dois vértices de um grafo, sendo ele utilizado no grafo bipartido para encontrar o caminho de custo mínimo passando somente por um número par de arestas. Além disso, para otimizar o algoritmo, não foram inseridas as arestas de peso ímpar nos grafos, visto que elas não são utilizadas e podem ser descartadas.

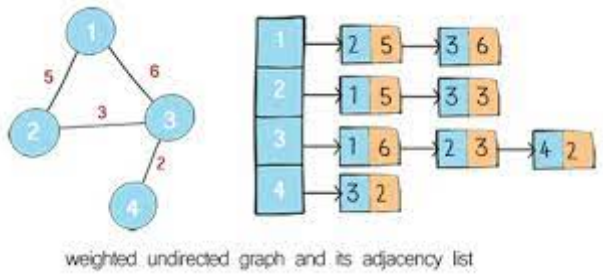
A solução funciona porque no grafo bipartido os vértices criados v0 e v1 representam conjuntos de número par e número ímpar, respectivamente. Dessa forma, todos os caminhos do grafo bipartido que terminam em v0 passam por um número par de arestas, pois ao percorrer uma aresta ele muda entre o conjunto de números pares e de números ímpares.



Dessa forma, o algoritmo retorna o caminho de custo mínimo passando somente por arestas pares e por uma quantidade par de arestas. Quando esse caminho não existe, o algoritmo retorna infinito (pois a distância é infinita), imprimindo “-1”.

3.1. Estruturas de Dados

Ao armazenar os grafos, foi utilizada uma lista de adjacência, uma forma simples de representar um grafo como uma lista de vértices, em que cada vértice tem uma lista de seus vértices adjacentes, e o peso da aresta que ligam até eles, como segue o exemplo da imagem a seguir:

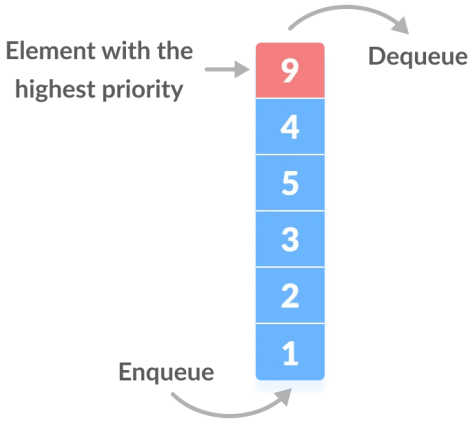


weighted undirected graph and its adjacency list

Para representar a lista de adjacência, foi utilizada a biblioteca padrão de C++ <vector>, de forma que foi armazenado um “vector de vector de pares”, ou seja, um vetor que armazena os vértices, e para cada vértice um vetor que armazena um par de números inteiros, para representar o vértice adjacente e o peso da aresta entre eles, respectivamente.

A lista de adjacência do grafo bipartido foi utilizando a mesma estrutura, porém com o dobro de tamanho, visto que o grafo bipartido gerado teria o dobro de vértices e o dobro de arestas do grafo original.

Ademais, ao rodar o dijkstra, foi utilizada uma fila de prioridades (priority_queue, da biblioteca queue de C++), que, sempre que um novo vértice não visitado é descoberto, ele é adicionado à fila de prioridade, e sempre que é terminado de observar todos os vizinhos de um vértice, ele é marcado como visitado e retirado da fila de prioridade. A fila de prioridade pode ser vista visualmente pelo exemplo da imagem a seguir:



3.2. Classes

A resolução do problema possui somente uma classe, a classe Graph. A classe graph possui como atributos e funções:

- numVertices: valor inteiro que armazena o número de cidades do problema, que representa o número de vértices do grafo.
- adjList: vetor que armazena outro vetor, e esse segundo vetor armazena um par de inteiros que representam as cidades vizinhas e a distâncias entre elas, ou seja, o segundo vetor na prática armazena um par de inteiros que representa os vértices adjacentes e o peso das arestas entre esses dois vértices.
- adjListBipartite: representa a mesma coisa que a adjList, mas para o grafo bipartido, então possui o dobro do tamanho (2n, sendo n o número de vértices do grafo).
- Graph(numVertices): construtor do grafo original, que inicializa os atributos e redefine o tamanho da lista de adjacência para a quantidade de vértices do grafo.
- addEdge(origin, destiny, distance): Adiciona os vértices iniciais, de destino e o peso da aresta do grafo original na lista de adjacência.
- newGraph(): Cria um novo grafo bipartido a partir do grafo original, duplicando a quantidade de vértices e arestas e mantendo uma correspondência entre as posições dos vértices nos grafos original e bipartido.
- dijkstra(x): Acha a distância com menor peso entre o vértice original e o vértice final. Nesse caso, como é rodado no grafo bipartido, encontra a distância com menor peso entre os vértices inicial e final passando somente por arestas de peso par e por um número par de arestas, conforme a especificação do problema.

4. Análise de Complexidade

4.1. Tempo

Começando a análise para as operações realizadas, e tomando como primeira operação a inserção de um grafo como lista de adjacência, temos que a complexidade de tempo para tal é linear, visto que o loop de inserção ocorre até o número de arestas, e realiza apenas operações de inserção, então possui complexidade O(n).

Para a função de criar o grafo bipartido, temos dois *for* aninhados, onde o externo executa V vezes (sendo V o número de arestas) e o interno executa e vezes (sendo e o número de arestas), e as demais operações de inserção constantes. Logo, temos que a complexidade da função que cria o grafo bipartido é O(V*E).

Por fim, para a função Dijkstra a complexidade é de O(V*E) , visto que o loop externo é executado uma quantidade V vezes e o loop interno é executado uma quantidade E vezes para cada execução de V.

4.2. Espaço

Analisando agora a complexidade de espaço e seguindo a mesma ordem da complexidade de tempo, temos, para a inserção do grafo na lista de adjacência, é ocupado um vetor de complexidade O($V + E$), visto que será armazenado no vetor os vértices e o par vertice-arestas.

Para a segunda função, criar o grafo bipartido, a complexidade espacial é O($2V + 2E$), pois duplica o grafo original, ou seja, duplica a quantidade de vértices e a quantidade de arestas.

Na última função, o Dijkstra, a complexidade de espaço é O(V), visto que a constante é ignorada, pois a função armazena o vetor de distâncias, um vetor de visitados e a fila de prioridade.

5. Referências

https://www.geeksforgeeks.org/graph-and-its-representations/
http://courses.csail.mit.edu/6.006/spring10/psets/ps5/ps5sol.pdf
https://www.lavivienpost.net/weighted-graph-as-adjacency-list/
https://www.programiz.com/dsa/priority-queue
https://www.geeksforgeeks.org/shortest-path-with-even-number-of-edges-from-source-to-destination/amp/