Representação de Grafos - Assignment 1

Diogo Araujo Miranda[1]

[1]ICEI - Pontifícia Universidade Católica de Minas Gerais (PUC-MG)

1. Contexto

Quando pensando em uma representação computacional para uma estrutura abstrata, devemos levar em conta tanto a quantidade de memória que irá utilizar - já que não possuímos ainda ”memória infinita” e também a ordem de complexidade de suas operações. Este documento busca justificar e demonstrar possíveis implementações utilizando estruturas já discutidas em literaturas, sendo estas matrize de adjacência e lista de adjacência, levando em consideração um certo caso. Todo código encontra-se no repositório: https://github.com/Diogo-Miranda/grafos/tree/main/TP01.

2. Grafo Direcionado Não Ponderado

Um grafo direcionado não ponderado é um conjunto de vértices em que um vértice u têm referência para outro vértice v por meio de uma aresta direcionada, e pode ser representado pela seguinte estrutura:

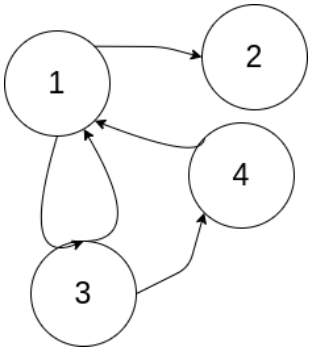

Figure 1. Grafo Direcionado Não Ponderado

Ao buscarmos referências de implementações na literatura, temos várias opções e suas vantagens. Neste trabalho foi utilizada a implementação de **matriz de adjacências** para um grafo direcionado não ponderado. Um dos principais motivos foi o custo de armazenamento, já que ao representar a matriz foi utilizado o tipo Boolean para representar a presença ou não de uma aresta entre dois vértices (pelo fato de não serem ponderados). Porém ainda têm-se a necessidade de alocar inicialmente todo o espaço que será utilizado pelo grafo, ou seja $O(N^2)$ de custo de uso de memória, onde N é o número de vértices. Por outro lado, têm-se um ganho em complexidade de implementação, já que é simples em termo de código representar uma matriz de adjacência e seus métodos. O custo em complexidade de busca é bastante satisfatório comparando-se com a lista de adjacências, em que para descobrir uma referência entre dois vértices teremos um custo de $O(1)$, já que realizaríamos a busca em matriz.

Uma possível codificação para a estrutura proposta em C++ poderia se resolvida com uma classe e alguns métodos, como por exemplo:

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

Figure 2. Matriz de Adjacências - Grafo Direcionado não ponderado

```
class Graph {
    public:
        bool matrizAdjacencia[NUM_MAX_VERTEX][NUM_MAX_VERTEX];
        Graph();
        void insert(int labelVertexOne, int labelVertexTwo);
        void print();
};
```

O número de vértices, como descrito, nos diz a dimensão de uma matriz a ser alocada, sendo informada previamente. Um dos seus métodos implementados é o método de inserir um vértice:

```
void Graph::insert(int labelVertexOne, int labelVertexTwo) {
    if(labelVertexOne > NUM_MAX_VERTEX || labelVertexOne <= 0 ||
        labelVertexTwo > NUM_MAX_VERTEX || labelVertexTwo <= 0)
       {
        printf("These vertex do not exists (%i, %i)\n",
           labelVertexOne, labelVertexTwo);
    } else {
        if(matrizAdjacencia[labelVertexOne][labelVertexTwo] ==
           1) {
            printf("There are edges between %i and %i",
               labelVertexOne, labelVertexTwo);
        } else {
            matrizAdjacencia[labelVertexOne][labelVertexTwo] =
               1;
        }
    }
}
```

3. Grafo Não Direcionado Não Ponderado

Nesse tipo de grafo, temos o mesmo desafio de representa-lo em uma estrutura de dados de faça sentido. A diferença de um grafo não direcionado é a questão do direcionamento, podendo ser representado por:

O tipo de estrutura de dados pensado foi também a questão da matriz de adjacências pelos mesmo motivos anteriores, já que iremos utilizar um tipo que utiliza apenas 1 bit para representar uma aresta. A diferença é que não iremos extrair um bom desempenho de armazenamento, sendo que teremos reflexão na diagonal principal:

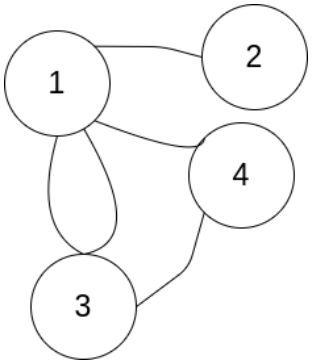

Figure 3. Matriz de Adjacências - Grafo Não Direcionado não ponderado

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

Figure 4. Grafo Não Direcionado não ponderado

Sua implementação é muito semelhante ao código demonstrado anteriormente, porém com uma diferença durante a inserção de um vértice, em que teremos a reflexão na diagonal principal:

```
1  void Graph::insert(int labelVertexOne, int labelVertexTwo) {
2      if(labelVertexOne != labelVertexTwo) {
3          if(labelVertexOne > NUM_MAX_VERTEX || labelVertexOne <=
               0 || labelVertexTwo > NUM_MAX_VERTEX ||
               labelVertexTwo <= 0) {
4              printf("These vertex do not exists (%i, %i)\n",
                   labelVertexOne, labelVertexTwo);
5          } else {
6              if(matrizAdjacencia[labelVertexOne][labelVertexTwo]
                   == 1) {
7                  printf("There are edges between %i and %i",
                       labelVertexOne, labelVertexTwo);
8              } else {
9                  matrizAdjacencia[labelVertexOne-1][
                       labelVertexTwo-1] = 1;
10                 matrizAdjacencia[labelVertexTwo-1][
                       labelVertexOne-1] = 1;
11             }
12         }
13     } else {
14         matrizAdjacencia[labelVertexOne-1][labelVertexOne-1] =
               1;
```

```
    }
}
```

4. Grafo Direcionado Ponderado

Nesse modelo de grafo, foi preferível utilizar a representação por meio da estrutura de lista de adjacências. Um dos principais motivos foi a facilidade em se ordenar um grafo por seu peso e futuramente ser utilizado em buscas que utilizam peso como base. A abordagem da matriz de adjacências não foi escolhida pelo motivo de que agora a representação das arestas não seria por um boolean, e sim por um inteiro ou um número de ponto flutuante. A representação segue a seguinte ideia:

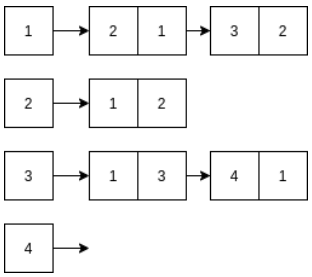

Figure 5. Grafo Direcionado Ponderado - Lista de Adjacências

Nessa representação cada lista nos representa uma referência no grafo direcionado, e cada nó cabeça representa um vértice que partem as referências. Cada cédula posterior nos indica uma aresta entre o vértice referente e seu peso. A principal vantagem é poder utilizar listas ordenadas em métodos de busca e também de organização do grafo em questão. Porém, em termos de percorrer o grafo teríamos um custo linear parar percorrer a lista de O(N), em que N representa a quantidade de vértices que um grafo tem. Seu código em C++ pode ser implementado como:

```
class Graph
{
    public:
        List *listaAdjacencia[NUM_MAX_VERTEX];
        Graph();
        void insertVertex(int label);
        void insertEdge(int labelVertexOut, int labelVertexIn,
           int weight);
        void print();

    private:
        void insert(Cell *vertex);

};

class List
{
    public:
        Cell *head;
```

```
19          List();
20          List(Cell *vertex);
21          List(int vertex, int weight);
22  };
23
24  class Cell
25  {
26      public:
27          int vertex;
28          int weight;
29          Cell *next;
30          Cell(int vertex, int weight, Cell *next);
31          Cell(int vertex, int weight);
32          Cell();
33          void print();
34  };
```

Nesse caso, utilizamos uma estrutura de lista simples com uma cédula cabeça, que nos indica o grafo de partida. A implementação de seu método de inserir poder ser feita da seguinte forma:

```
1   void Graph::insertVertex(int label) {
2       Graph::insert(new Cell(label, 0));
3   }
4
5   void Graph::insert(Cell *vertex) {
6
7       int index = (vertex->vertex)-1;
8
9       if(listaAdjacencia[index] == NULL) {
10          listaAdjacencia[index] = (List*)malloc(sizeof(List));
11          listaAdjacencia[index] = new List(vertex);
12      } else {
13          printf("These vetex exists");
14      }
15  }
16
17  void Graph::insertEdge(int labelVertexOut, int labelVertexIn,
        int weight) {
18      int index = labelVertexOut-1;
19
20      if(listaAdjacencia[index] != NULL && listaAdjacencia[
            labelVertexIn-1] != NULL) {
21          Cell *aux = listaAdjacencia[index]->head;
22
23          while(aux->next != NULL) {
24              aux = aux->next;
25          }
26
27          aux->next = new Cell(labelVertexIn, weight);
```

```
28      }
29  }
```

5. Grafo Não Direcionado Não Ponderado

Seguindo a mesma linha de raciocínio para o grafo direcionado, nesse tipo de representação de um grafo, utilizamos uma lista de adjacência. Cada cédula da lista indica um vértice juntamente com seu peso. Nessa representação no grafo direcionado, ganhamos em termo de poder calcular um peso personalizado em casa relação entre grafos, além de poder realizar buscas em listas ordenadas.

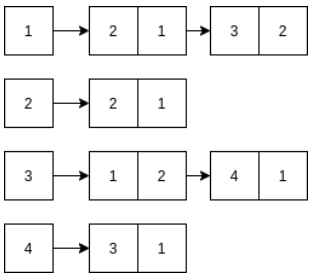

Figure 6. Grafo Não Direcionado Ponderado - Lista de Adjacências

Sua implementação em C++ tem uma pequena diferença em comparação ao direcionado, já que no método de inserção devemos ter uma referência também para o vértice de entrada:

```
1  void Graph::insertVertex(int label) {
2      Graph::insert(new Cell(label, 0));
3  }
4
5  void Graph::insert(Cell *vertex) {
6
7      int index = (vertex->vertex)-1;
8
9      if(listaAdjacencia[index] == NULL) {
10         listaAdjacencia[index] = (List*)malloc(sizeof(List));
11         listaAdjacencia[index] = new List(vertex);
12     } else {
13         printf("These vetex exists");
14     }
15 }
16
17 void Graph::insertEdge(int labelVertexOut, int labelVertexIn,
       int weight) {
18     int indexOut = labelVertexOut-1;
19     int indexIn = labelVertexIn-1;
20
21     if(listaAdjacencia[indexOut] != NULL && listaAdjacencia[
           indexIn] != NULL) {
22         Cell *aux = listaAdjacencia[indexOut]->head;
23
```

```
24          // Insert reference to "vertex Out"
25          while(aux->next != NULL) {
26              aux = aux->next;
27          }
28
29          aux->next = new Cell(labelVertexIn, weight);
30
31          // Insert reference to "vertex In"
32          aux = listaAdjacencia[indexIn]->head;
33
34          while(aux->next != NULL) {
35              aux = aux->next;
36          }
37
38          aux->next = new Cell(labelVertexOut, weight);
39      }
40  }
```