Sumário

Busca em Largura e Profundidade Para Grafos Com Lista de Adjacência, Matriz de Adjacência e Matriz de Incidência

Grafos

Formalmente, um grafo é uma colecção de *vértices* (*V*) e uma colecção de *arcos* (*E*) constituídos por pares de vértices.
Pense nos vértices como "*locais*". O conjunto dos vértices é o conjunto de todos os locais possíveis. Nesta analogia, os arcos (ou arestas) representam caminhos entre estes locais. O conjunto *E* (vou usar o termo mais comum – "*E*" do inglês "*edges*") contém todas as ligações entre os locais.
Utilizar grafos é de grande utilidade na representação de problemas da vida real.

Podem ser cidades, e uma rede de estradas. Redes de computadores. Até mesmo os movimentos de um cavalo num tabuleiro de xadrez podem ser representados através de um grafo.

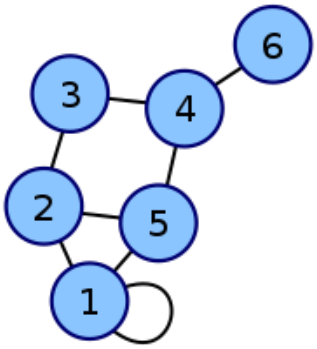
Para representar um grafo de forma computacional, iremos abordar três maneiras de desenvolver o código, sendo elas:

Matriz de Adjacência

Dado um grafo *G* com *n* vértices, podemos representá-lo em uma matriz *n* x *n* $A(G)=[a_{ij}]$ (ou simplesmente *A*). A definição precisa das entradas da matriz varia de acordo com as propriedades do grafo que se deseja representar, porém de forma geral o valor a_{ij} guarda informações sobre como os vértices v_i e v_j estão relacionados (isto é, informações sobre a adjacência de v_i e v_j).

Para representar um grafo não direcionado, simples e sem pesos nas arestas, basta que as entradas a_{ij} da matriz *A* contenham 1 se v_i e v_j são adjacentes e 0 caso contrário. Se as arestas do grafo tiverem pesos, a_{ij} pode conter, ao invés de 1 quando houver uma aresta entre v_i e v_j, o peso dessa mesma aresta.

Desta maneira, temos o grafo e a matriz correspondente nas imagens a seguir.



$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Matriz de Incidência

Uma **matriz de incidência** representa computacionalmente um grafo através de uma matriz bidimensional, onde uma das dimensões são vértices e a outra dimensão são arestas.

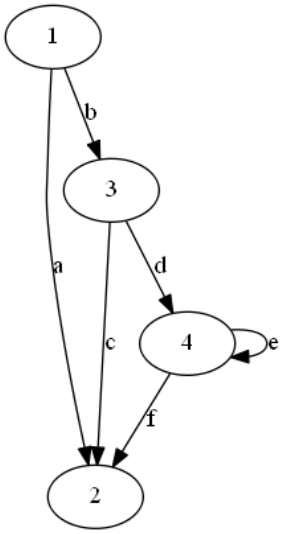
Dado um grafo G com *n* vértices e *m* arestas, podemos representá-lo em uma matriz *n* x *m* M. A definição precisa das entradas da matriz varia de acordo com as propriedades do grafo que se deseja representar, porém de forma geral guarda informações sobre como os vértices se relacionam com cada aresta (isto é, informações sobre a *incidência* de uma aresta em um vértice[1]).

Para representar um grafo sem pesos nas arestas e não direcionado, basta que as entradas da matriz M contenham 1 se a aresta incide no vértice, 2 caso seja um laço (incide duas vezes) e 0 caso a aresta não incida no vértice.

Desta maneira, temos o grafo e a matriz correspondente nas imagens a seguir.

	a	b	c	d	e	f
1	1	1	0	0	0	0
2	1	0	1	0	0	1
3	0	1	1	1	0	0
4	0	0	0	1	2	1

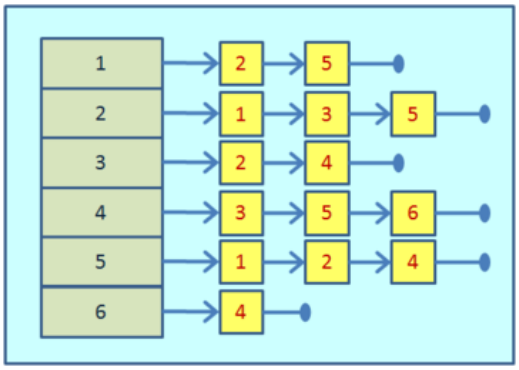
$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 \end{bmatrix}$$



Lista de Adjacência

lista de adjacência é uma estrutura de dados para representar grafos. Em uma representação de lista de adjacência, podemos manter, para cada vértice do grafo, uma lista de todos os outros vértices com os quais ele tem uma aresta (a "lista de adjacência", deste vértice). Por exemplo, a representação sugerida por van Rossum, em que uma tabela de dispersão (*tabela hash*) é usada para associar cada vértice com um array de vértices adjacentes, pode ser vista como um exemplo deste tipo de representação. Outro exemplo é a representação encontrada em Cormen et al. em que um array indexado pelos números dos vértices aponta para uma lista simplesmente encadeada dos vizinhos de cada vértice.

Veja um exemplo a seguir.



Para podermos navegar nesta estrutura, iremos apresentar dois métodos de busca, sendo eles:

Busca em Largura (BFS)

Na teoria dos grafos, **busca em largura** (ou busca em amplitude, também conhecido em inglês por Breadth-First Search - BFS) é um método de busca não-informada (ou desinformada) que expande e examina sistematicamente todos os vértices de um grafo direcionado ou não-direcionado. Em outras palavras, podemos dizer que o algoritmo realiza uma busca exaustiva num grafo passando por todas as arestas e vértices do grafo. Sendo assim, o algoritmo deve garantir que nenhum vértice ou aresta será visitado mais de uma vez e, para isso, utiliza uma estrutura de dados fila para garantir a ordem de chegada dos vértices. Dessa maneira, as visitas aos vértices são realizadas através da ordem de chegada na estrutura fila e um vértice que já foi marcado não pode entrar novamente a esta estrutura.

Basicamente, ela realizar uma busca ou travessia num grafo e estrutura de dados do tipo árvore. Intuitivamente, você começa pelo vértice raiz e explora todos os vértices vizinhos. Então, para cada um desses vértices mais próximos, exploramos os seus vértices vizinhos inexplorados e assim por diante, até que ele encontre o alvo da busca.

Busca em Profundidade (DFS)

Na teoria dos grafos, **busca em profundidade** (ou busca em profundidade-primeiro, também conhecido em inglês por Depth-First Search - DFS) é um algoritmo usado para realizar uma busca ou travessia numa árvore, estrutura de árvore ou grafo. Intuitivamente, o algoritmo começa num nó raiz (selecionando algum nó como sendo o raiz, no caso de um grafo) e explora tanto quanto possível cada um dos seus ramos, antes de retroceder(backtracking).

Formalmente, um algoritmo de busca em profundidade realiza uma busca não-informada que progride através da expansão do primeiro nó filho da árvore de busca, e se aprofunda cada vez mais, até que o alvo da busca seja encontrado ou até que ele se depare com um nó que não possui filhos (nó folha). Então a busca retrocede (backtrack) e começa no próximo nó.

Abordagem do Código

A aplicação desenvolvida e presente nesta página, se trata de três arquivos com abordagens diferentes sobre a implementação de grafos, abordagem do ponto de vista dinâmico (Lista de Adjacência) e estático (Matriz de Adjacência e Matriz de Incidência). Para desenvolver tais aplicações foi pensado um grafo não direcional em que não há ocorrências de Laços.

Estruturas Básicas

```
typedef struct graph *Graph;
typedef struct TipoVertex *Vertex;
struct TipoVertex{
    int value;
    Vertex prox;
};
struct graph{
    int numbVert;
    int numbEdges;
    Vertex *adj;
};
```

Temos a primeira estrutura abordada representativa de grafos que são as listas de adjacência, no qual, o 'TipoVertex' representa uma posição da lista que guarda o valor e o endereço do próximo item, e, o 'graph' representa o Grafo em se em que temos o número de vértices do grafo (numbVert), o número de arestas (numbEdges) e por fim um ponteiro que guardara os vértices com suas conexões.

```
typedef struct graph *Graph;

struct graph{
    int numbVert;
    int numbEdges;
    int** adj;
};
```

A próxima é por Matriz de Adjacência em que não precisamos mais do Vertex, pois teremos somente uma matriz quadrática que guardará todas as informações de relação dos vértices

```
typedef struct graph *Graph;

struct graph{
    int numbVert;
    int numbEdges;
    int numbEdgesAtual;
    int** adj;
};
```

E por fim, temos a Matriz de incidência, em que foi incluída apenas uma variável (numbEdgesAtual) para identificar quantas arestas já foram adicionadas, visto que a já existente (numbEdges) agora marca quantas arestas totais o grafo tem.

BFS (Código)

A necessidade de mudança no código BFS se concentra na maneira de caminhar dentro do grafo (while).

Segue abaixo o BFS padrão desenvolvido para estruturas de Lista de Adjacência, em que vertex sempre caminha até o final da lista de adjacência de cada vértice.

```
while(fila->size > 0)
{
    Item *u = Dequeue(fila);
    for(Vertex vertex = graph->adj[u->data]; vertex != NULL; vertex = vertex->prox)
    {
        if(color[vertex->value] == 0)
        {
            color[vertex->value] = 1;
            distance[vertex->value] = distance[u->data] + 1;
            father[vertex->value] = u->data;
            Queue(fila,vertex->value);
        }
    }
    color[u->data] = 2;
    cout<<"Vertex: "<<u->data<<endl;
}
```

Visto o código acima, para torna-lo funcional para um grafo de matriz de adjacência, basta começar a considerar somente os valores diferente de 0 na matriz quadrática.

```
while(fila->size > 0)
{
    Item *u = Dequeue(fila);
    for(int vertex = 0; vertex < graph->numbVert; vertex++)
    {
        if(color[vertex] == 0 && graph->adj[i][vertex] != 0)
        {
            color[vertex] = 1;
            distance[vertex] = distance[u->data] + 1;
            father[vertex] = u->data;
            Queue(fila,vertex);
        }
    }
    i++;
    color[u->data] = 2;
    cout<<"Vertex: "<<u->data<<endl;
}
```

E por fim, para modificar o código pensando na estrutura de matriz de incidência, temos que levar em consideração que as colunas da matriz agora representam arestas (que podem conectar somente dois vértices), o que implica que será necessário fazer uma busca na coluna para encontra os dois vértice que esta conectados na mesma aresta e por fim, dar sequência no código DFS. Para isso, foi incluído um for que após encontra uma ponta da aresta, procura pela outra.

```
while(fila->size > 0)
{
    Item *u = Dequeue(fila);
    for(int vertex = 0; vertex < graph->numbEdges; vertex++)
    {
        if(graph->adj[i][vertex] != 0)
        {
            for(int k = 0; k < graph->numbVert; k++)
            {
                if(color[k] == 0 && graph->adj[k][vertex] != 0 && k != i)
                {
                    color[k] = 1;
                    distance[k] = distance[u->data] + 1;
                    father[k] = u->data;
                    Queue(fila,k);
                    break;
                }
            }
        }
    }
    i++;
    color[u->data] = 2;
    cout<<"Vertex: "<<u->data<<endl;
}
```

DFS (Código)

Como no BFS as únicas mudanças que foram necessárias no código, foram no modo de caminhamento dentro do grafo (for) que varia para cada estrutura. Veja abaixo essas mudanças na prática, que seguem uma lógica similar das mudanças feitas no tópico passado

```
for(Vertex u = graph->adj[vertex->value]; u != NULL; u = u->prox)
    if(color[u->value] == 0)
        DFS_VISIT(graph, u, color, distance, finale, time);
```

```
for(int u = 0; u < graph->numbVert; u++)
    if(color[u] == 0 && graph->adj[vertex][u] != 0)
        DFS_VISIT(graph, u, color, distance, finale, time);
```

```
for(int u = 0; u < graph->numbEdges; u++)
{
    if(graph->adj[vertex][u] != 0)
    {
        for(int k = 0; k < graph->numbVert; k++)
        {
            if(color[k] == 0 && graph->adj[k][u] != 0 && k != vertex)
            {
                DFS_VISIT(graph, k, color, distance, finale, time);
                break;
            }
        }
    }
}
```

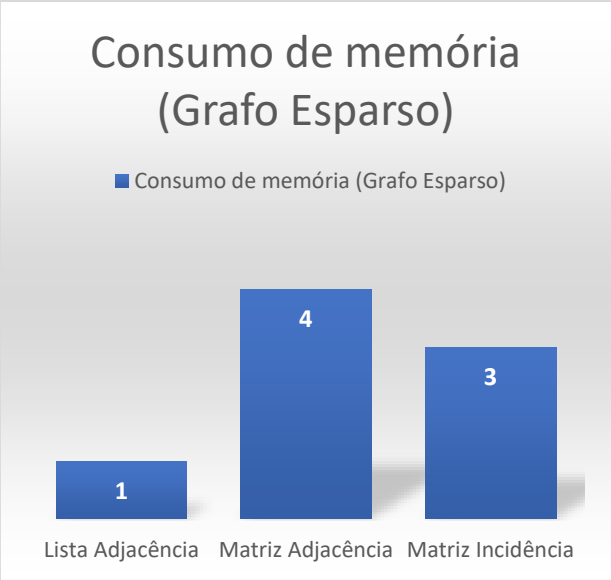
Tempo de Execução

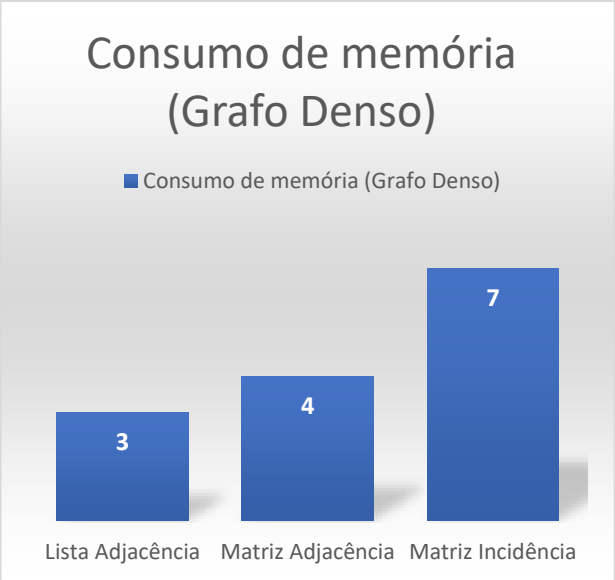
Para obter os seguintes resultados, foi feito 5 testes para cada método de implementação utilizando os mesmos grafos.

Tempo de Execução (s)	ESPARSO		DENSO	
	BFS	DFS	BFS	DFS
Lista Adjacência	0.0516	0.0621	0.0553	0.0673
Matriz Adjacência	0.0492	0.0605	0.0504	0.0663
Matriz Incidência	0.0834	0.0877	0.0856	0.0893

Consumo de Memória

Fazendo uma analise da implementação da estrutura, intuitivamente podemos perceber que há custos diferentes (tanto para análise de tempo quanto para análise de memória) para as diferentes maneiras de implementação. Veja a seguir os seguintes gráficos simbolizando o consumo de memória em cada estrutura:





Conclusão

Visto os custos de memória e tempo de cada algoritmo, podemos escolher um melhor para fazer nossas implementações. O de matriz de incidência tem um tempo de execução bem elevado comparado a outras estruturas, visto que seu gasto de memória em um grafo esparso se torna um pouco menor do que o da matriz de adjacência, mas em um grafo denso se torna maior, nos permite dizer que essa não é uma implementação muito boa para ser usada em grandes projetos. Já por matriz de adjacência, o tempo de execução se mostra o melhor da comparação, mas tem um custo de memória que será de forma simplificada v^2, o que deixa evidente seu alto consumo de memória de forma desnecessária em caso de grafos com muitos vértices e poucas arestas. Por fim tempo a implementação por Lista de adjacência que nos dá um tempo de execução próximo ao da matriz de adjacência (que na comparação é o melhor) e tem o menor consumo de memória em de todos os casos abordados, tornando esta, a melhor solução tanto para problemas menores, quanto para problemas maiores.

Referências

https://www.revista-programar.info/artigos/grafos-1a-parte/

https://pt.wikipedia.org/wiki/Matriz_de_adjac%C3%AAncia

https://pt.wikipedia.org/wiki/Matriz_de_incid%C3%AAncia#:~:text=Uma%20matriz%20de%20incid%C3%AAncia%20representa,matriz%20n%20x%20m%20M.

https://pt.wikipedia.org/wiki/Lista_de_adjac%C3%AAncia

https://pt.wikipedia.org/wiki/Busca_em_largura

https://pt.wikipedia.org/wiki/Busca_em_profundidade