Programação I

Estruturas de dados não lineares
(Grafos)

Estrela Ferreira Cruz

1

Objetivos da aula

Estruturas de Dados não Lineares – grafos:

- Constituição de um grafo;
- Apresentação dos principais conceitos sobre grafos;
- Representação dos grafos:
  - Matriz de adjacência;
  - Listas de adjacência;
- Algoritmos de travessias d e grafos:
  - BFS (travessia em largura);
  - DFS (travessia em profundidade);
- O Algoritmo de procura do caminho mais curto: Dijkstra

2

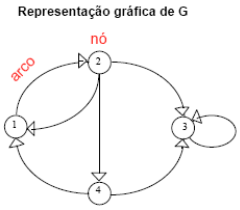
2

Grafos

Estruturas de dados não lineares: árvores e grafos

- As **estruturas de dados não lineares** permitem representar informação não linear na forma hierárquica (árvores) e **na forma de rede (grafos).**
- Um grafo é um **conjunto de pontos**, chamados **vértices** ou nós, **conectados por linhas**, chamadas de **arcos** ou **arestas** (ou Edges).
- **Um vértice** é um objeto simples que pode ter nome e outros atributos.

Exemplo:



G = (V, A) em que:
V = {1, 2, 3, 4}
A = {(1,2), (2,1), (2,3), (2,4), (3,3), (4,1), (4,3)}

3

3

Grafos

Dependendo da aplicação:

- **As arestas podem, ou não, ter direção**. Se as arestas tiverem uma direção associada (indicada por uma seta na representação gráfica) temos um **grafo direcionado**, ou dígrafo.
- **Vértices e/ou arestas podem ter um peso** (numérico) associado.

Tipos de grafos:

- **Grafo nulo** é o grafo cujos conjuntos de vértices e de arestas são vazios.
- **Grafo trivial** ou "ponto" é um grafo com um único vértice e sem arestas.
- **Grafo orientado** é um grafo que contém **arcos com um único sentido**
- **Grafo não orientado** é um grafo que contém arcos que podem ser percorridos nos **dois sentidos**.
- **Grafo misto** – contém arcos dos dois tipos anteriores

4

4

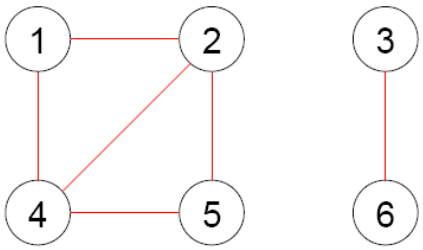
Grafos

Grafo não orientado

Num grafo não orientado o conjunto A é constituído por pares não ordenados (conjuntos com 2 elementos). Assim, (i, j) e (j, i) correspondem ao mesmo arco.

No exemplo temos: V = {1, 2, 3, 4, 5, 6},

A = {(1, 2), (1, 4), (2, 4), (2, 5), (3, 6), (4, 5)}



5

5

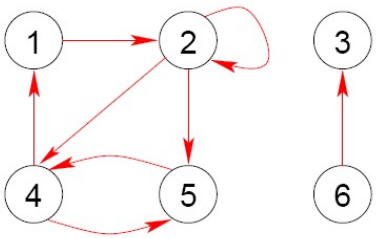
Grafos

Grafo orientado

Um grafo orientado é um par (V,A) com V um conjunto finito de vértices ou nós e A (Arestas) uma relação binária em V – o conjunto de arcos ou arestas do grafo.

Exemplo: V = {1, 2, 3, 4, 5, 6},

A = {(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)}

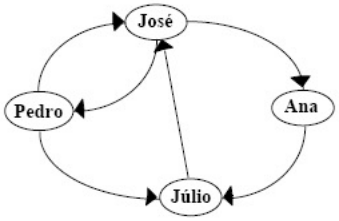


6

6

Grafos

- Grafo completo – Um grafo completo de N nós é um grafo que contém sempre uma ligação entre todos os nós distintos.
- Grafo não conexo – Um grafo G é considerado não conexo quando existe um conjunto de nós dissociados ou desagregados.
- Grafo conexo – Um grafo G é conexo se dados quaisquer nós adjacentes V e W existe sempre uma ligação entre o nó V e o nó W. Um exemplo de um grafo conexo é o grafo da figura seguinte:

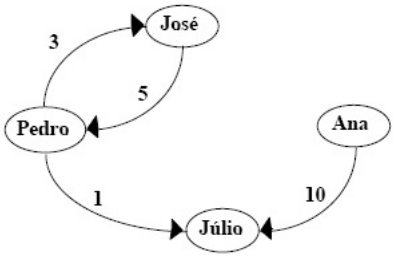


7

7

Grafos

Um grafo pode ter números, ou pesos associados a cada arco. Nesse caso o grafo chama-se pesado e o número junto a cada arco designa-se por peso do arco. Podemos ver um exemplo de um grafo pesado na figura que se segue.

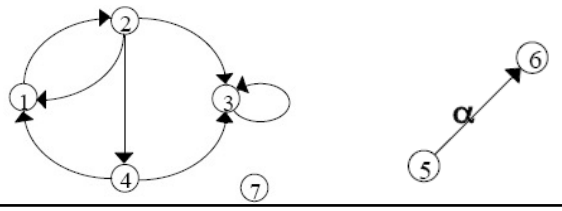


8

8

Grafos

- Um nó i é incidente a um arco x, se i é um dos dois nós no par ordenado que define x.
- Um arco que liga nós adjacentes diz-se incidente a esses nós.
- Um nó j é adjacente a um nó i se existe um arco de i para j. Nesse caso, i e j são respetivamente os vértices de origem e de destino do arco (i, j). No grafo da figura em baixo o nó 1 é adjacente de 2 e 4
- A relação de adjacência pode não ser simétrica num grafo orientado.
- Se j é adjacente a i, j é chamado de sucessor de i e i é o predecessor de j.
- Um nó pode não ter nenhum arco associado. Por exemplo o nó 7.

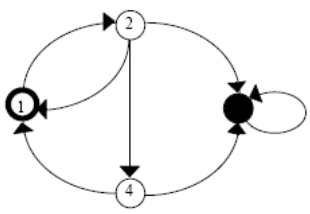


9

9

Grafos

- O grau de um nó é o número de arcos incidentes a ele.
- O grau de entrada de um nó num grafo orientado é o número de arcos com destino no nó, ou seja, o grau de entrada de um nó i é o número de arcos que têm i como segundo nó do par ordenado.
- O grau de saída de um nó num grafo orientado é o número de arcos com origem nesse nó, ou seja, é o primeiro nó do par ordenado.
- O grau do nó é a soma de ambos.
- Um nó de grau zero é dito isolado ou não conectado.



Para o nó 1 temos:
Grau entrada = 2
Grau saída = 1
Grau do nó = 3

10

10

Grafos

Caminho entre vértices

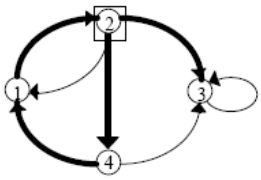
- Num grafo (V, A), um caminho do vértice x para o vértice y é uma sequência de vértices {v0, v1, . . . vk } tais que x= v0 e y= vk e (vi-1, vi) Є A, para i=1,2,..,k.
- Um caminho diz-se simples se todos os seus vértices são distintos.
- Se existir um caminho c de x a y então y é alcançável a partir de x via c.
- O comprimento de um caminho é o número de arcos nele contidos:

  (v0, v1), (v1, v2), . . . (vk−1, vk).
- Um subcaminho de um caminho é uma qualquer sua subsequência contígua.
- Existe sempre um caminho de comprimento 0 de um vértice para si próprio;
- Um ciclo é um caminho de comprimento >= 1 com início e fim no mesmo vértice.
- Um grafo diz-se acíclico se não contém ciclos.

11

11

Grafos



Em relação ao grafo do lado:

- É possível a partir do nó 1 atingir o nó 3 percorrendo os arcos (1,2) e (2,3). Estes arcos formam um caminho de 1 para 3.
- O comprimento do caminho, neste caso, é 2.
- Existe um ciclo que une os nós 1, 2 e 4.
- Logo este é um Grafos cíclico porque contêm pelo menos um ciclo.
- Um ciclo de um único arco (arco(i,i)) chama-se um anel ou laço. Caso do nó 3, arco (3,3) da figura.
- Os anéis são interditos nos grafos não orientados.
- Dois caminhos são independentes se não tiverem nenhum vértice em comum com exceção do primeiro e do último.

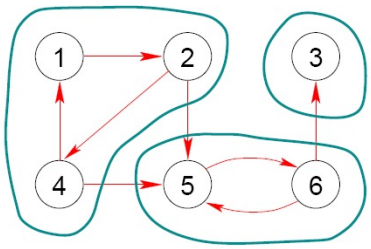
12

12

Grafos

Um grafo (V', E') é um sub-grafo de (V, E) sse $V' \subseteq V$ e $E' \subseteq E$.

- Um grafo não orientado diz-se ligado se para todo o par de vértices existe um caminho que os liga. Os componentes ligados de um grafo são os seus maiores subgrafos ligados.
- Um grafo orientado diz-se fortemente ligado se para todo o par de vértices (m, n) existem caminhos de m para n e de n para m.
- Os componentes fortemente ligados de um grafo são os seus maiores subgrafos fortemente ligados.



13

13

Grafos

Travessias: As duas formas mais usadas para percorrer os nós de um grafos são:

- através de uma travessia do grafo em profundidade (DFS - Depth First Search)
- através de uma travessia em largura (BFS - Breadth-First Search)

- Dado um grafo G = (V,A) e um seu vértice s, o algoritmo de pesquisa em largura ou BFS (Breadth-First Search):
- Produz uma árvore (subgrafo de G) com raiz s contendo todos os vértices alcançáveis a partir de s;
- Começa pela raiz (s) e explora todos os nós, ou seja todos os vértices alcançáveis a partir de s antes de avançar para o nível seguinte.
- Recorre ao uso de uma queue para armazenar os vértices visitados não tratados.
- Este algoritmo funciona para grafos orientados e não orientados.

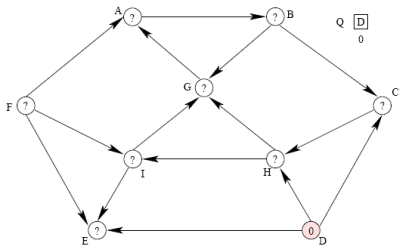
Note-se que uma árvore é sempre um grafo, mas um grafo pode não ser uma árvore.

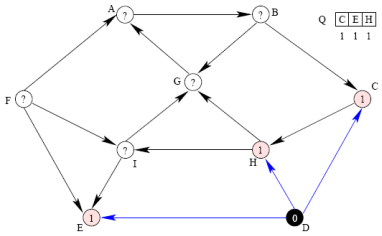
14

14

Grafos

Exemplo: Começando pelo vértice D (fica com o valor 0). Passa para os vértices C, E e H que ficarão numerados com 1 como se vê na 2º figura.





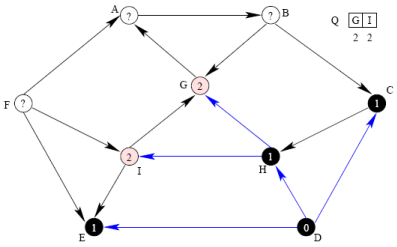
15

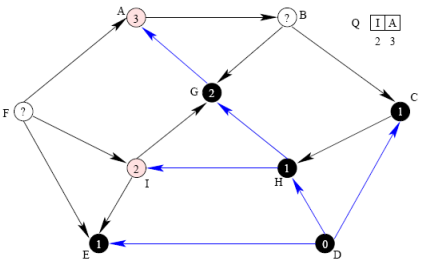
15

Grafos

Em seguida passa para os vértices G e I que ficarão numerados com 2 (fig. 3).

Em seguida passa para o vértice A que ficará numerado com 3 (fig. 4).



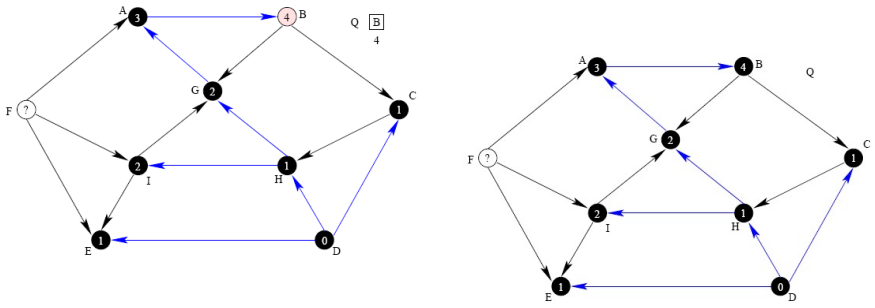


16

16

Grafos

Em seguida passa para o vértice B que ficará numerado com 4. Assim, a pesquisa fica completa. O vértice F não é alcançável a partir de D, logo o algoritmo não passa por este vértice. Em geral, num grafo não fortemente ligado, é necessário iniciar uma nova pesquisa em cada componente ligado, para garantir que todos os vértices são alcançados.



17

17

Grafos

Travessia em profundidade ou DFS (Depth First Search)

Este método tem sempre por base um dado nó de partida (inicial). O procedimento efetua uma pesquisa sistemática de nós adjacentes ao nó corrente. A ordem de visita dos nós depende do nó inicial e na escolha do nó seguinte a visitar tendo em conta os vários nós vizinho ainda por visitar.

Este algoritmo utiliza a seguinte estratégia para efetuar a travessia do grafo:

- Os próximos arcos a explorar têm origem no mais recente vértice descoberto que ainda tenha vértices adjacentes não explorados.
- Assim, quando todos os adjacentes a v tiverem sido explorados, o algoritmo recua ("backtracks") para explorar vértices com origem no nó a partir do qual v foi descoberto.
- Depois de terminada a pesquisa com origem em s, serão feitas novas pesquisas para descobrir os nós não alcançáveis a partir de s.
- O grafo dos antecessores de G é, neste caso, uma floresta composta de várias árvores de pesquisa em profundidade.

18

18

Grafos

Generalização da travessia em pré-ordem

- executada numa árvore - visita sistemática de todos os vértices;
- executada num grafo:
  - evitar os ciclos - marcar os nós visitados para impedir a repetição
  - ficam os vértices por visitar - percorrer lista de nós até ao seguinte não marcado

Uma pesquisa em profundidade a começar em qualquer nó, visita todos os nós.

- Ao processar (v, w), se w não estiver marcado acrescenta-se a aresta na árvore, se já estiverem ambos marcados, acrescenta-se uma aresta de retorno (a tracejado) que não pertence à árvore.
- Árvore simula a pesquisa; numeração em pré-ordem pelas arestas da árvore dá a ordem de marcação dos vértices (numerar antes de aceder a adjacentes)

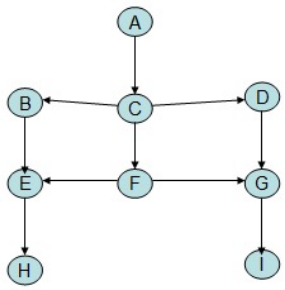
19

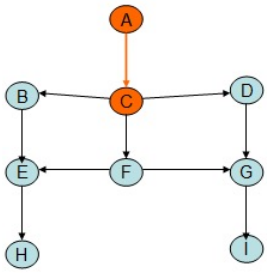
19

Grafos

Exemplo de execução :

- No grafo que se apresenta em baixo, vamos começar a travessia pelo vértice A.
- O único vértice adjacente a A é o vértice C, por isso continuamos a nossa travessia passando para o vértice C. Vamos sinalizar os nós já visitados com outra cor (laranja).



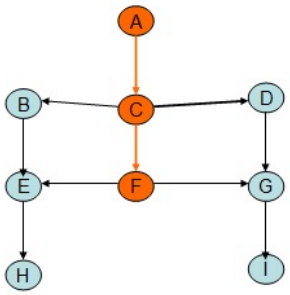


20

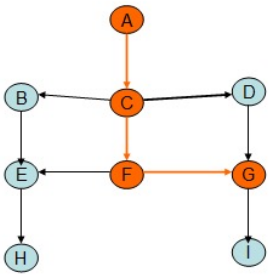
20

Grafos

O vértice C tem 3 vértices adjacentes: B,F e D. Vamos optar por selecionar o vértice com letra alfabeticamente mais alta: F.



De F e seguindo o critério anterior, passamos para G.

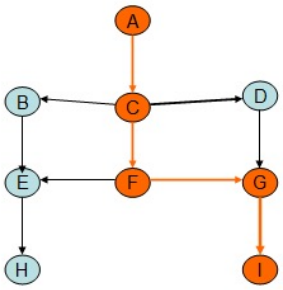


21

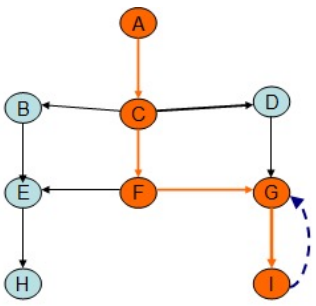
21

Grafos

G tem apenas um nó adjacente, o nó I, por isso, passamos para I.



I não tem nós adjacentes, por isso, temos que recuar (backtracking) e regressamos a G (representado a tracejado na figura).

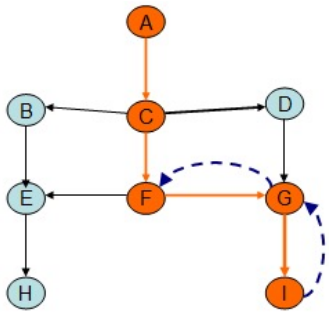


22

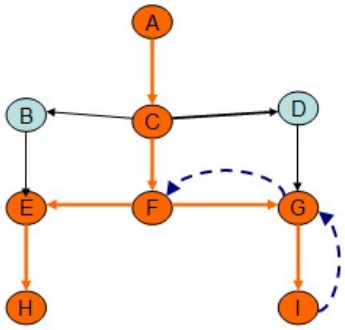
22

Grafos

Uma vez que todos os vértices adjacentes a G já foram visitados, recuamos para F.



O único vértice adjacente a F e ainda não visitado é o vértice E. Avançamos, então, para E e em seguida para H, porque é o único vértice adjacente a E.

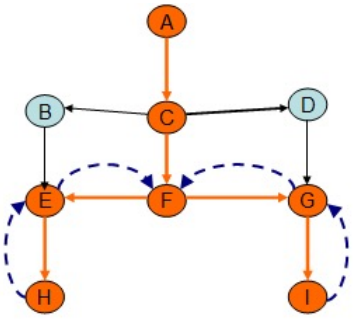


23

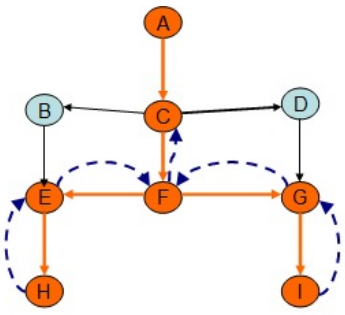
23

Grafos

H não tem vértices adjacentes, recuamos para E. Como o único vértice adjacente a E (H) já foi visitado, recuamos para F.



De F, uma vez que já todos os vértices adjacentes foram visitados, recuamos para C.

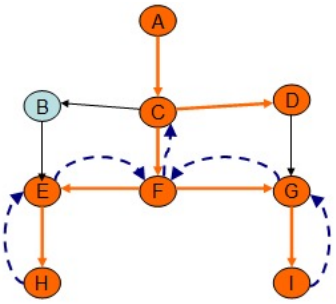


24

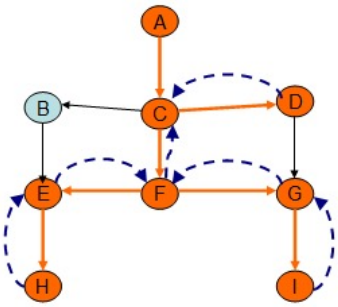
24

Grafos

Em C temos 2 vértices adjacentes não visitados: B e D. Optando pelo vértice alfabeticamente mais alto, vamos para D.



De D, temos cesso a G, mas já foi visitado, por isso recuamos novamente para C.

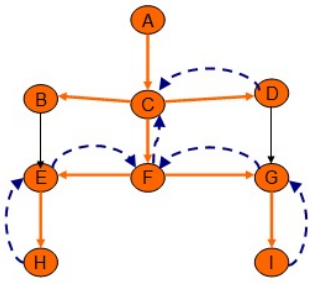


25

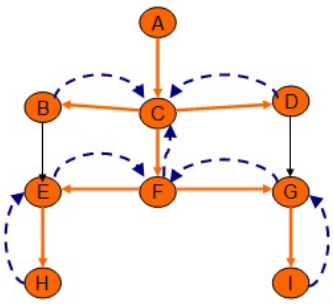
25

Grafos

C tem, neste ponto da travessia, um vértice adjacente ainda não visitado: o vértice B. Seguimos então para B.



O único vértice adjacente a B (E) já foi visitados, por isso, recuamos para C.

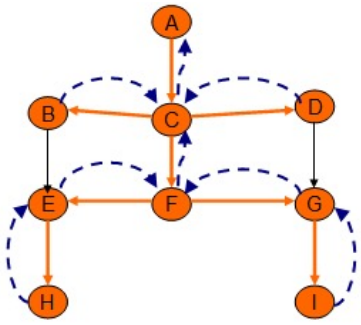


26

26

Grafos

Em C, todos os nós adjacentes já foram visitados, recuamos para A. Como o único vértice adjacente a A já foi visitado, finalizamos a pesquisa.



O resultado da travessia é o seguinte: A C F G I E H D B

27

27

Grafos – Matriz de adjacência

Representação de Grafos:

- Existem várias formas de representar grafos, usando para isso estruturas de dados já estudadas.
- Os grafos podem ser representados através de **matrizes ou através de listas ligadas**.

Representação Matricial:

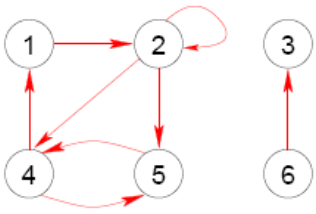
- Uma matriz de ligações adjacentes, ou **Matriz de adjacências**, é a forma mais simples de representar um grafo orientado.
- Trata-se de uma representação estática, é por isso apropriada para grafos densos, em que |A| (ou |E|) se aproxima de |V|2.

28

28

Grafos – Matriz de adjacência

Exemplo de uma matriz de adjacência:



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

Trata-se de uma matriz de dimensão $|V| \times |V|$, $A = (a_{ij})$ com $a_{ij} = 1$ se $(i, j) \in E$; $a_{ij} = 0$ em caso contrário.

29

29

Grafos – Matriz de adjacência

Implementação

A Matriz de adjacência pode ser representada da seguinte forma:

```
int adj[NumNos][NumNos];
```

onde NumNos representa o número de nós que constituem o grafo.

- Se grafo[i-1][k-1]==1 && grafo[k-1][j-1]==1 então temos um caminho, de comprimento 2, de i para j, passando por k.
- No caso de um grafo não orientado, a matriz de adjacências é simétrica, e é possível armazenar apenas o triângulo acima da diagonal principal.
- Se o grafo for pesado, o peso de cada arco pode ser incluído na respetiva posição da matriz, podendo ser representada pela mesma estrutura de dados.

30

30

Grafos – Listas de adjacência

Representação de Grafos:

Representações recorrendo ao uso de **listas ligadas**.

- Esta é a representação mais usual.
- Esta representação usa um conjunto de listas ligadas, chamadas **listas de adjacência**, e um **vetor para aceder a cada uma das listas**. Ou seja, consiste num **array de listas ligadas**.
- **Esta representação é útil** para grafos em que o numero de arcos |A| seja pequeno, menor que |V| 2 - **grafos esparsos**. (Numa matriz esparsa a maioria dos elementos é igual a zero, sendo apenas 30% dos valores, significativos).

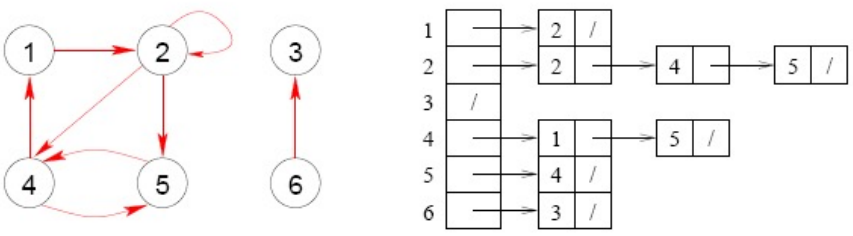
31

31

Grafos – Listas de adjacência

Representação de Grafos em listas se adjacência:

- A representação do seguinte grafo em lista de adjacência seria a seguinte:



32

32

Grafos

Algumas notas:

- A soma do comprimento das listas ligadas é o nº arcos (|A|).
- Se o grafo for pesado (i.e., se contiver informação associada aos arcos), o peso de cada arco pode ser incluído no nó respetivo numa das listas ligadas.
- No caso de um grafo não orientado, esta representação pode ser utilizada desde que antes se converta o grafo num grafo orientado, substituindo cada arco (não orientado) por um par de arcos (orientados).
- Neste caso a representação contém informação redundante e o comprimento total das listas é 2*|A|.
- Uma vantagem em relação às listas de adjacências é o facto de ser imediato verificar a adjacência de dois nós (sem ter que percorrer uma lista ligada).

33

33

Grafos – Listas de adjacência

A estruturas de dados pode ser a seguinte:

```
typedef int VERTICE;
typedef struct nodo {
     VERTICE v;
     struct nodo *next;
}NODO;
NODO *grafo[NumNos];
```

Para um grafo pesado o vértice pode ser representado pela seguinte estrutura:

```
Typedef struct {
     int adj; int peso;
}VERTICE;
```

As listas, bem como o array de listas, pode ser representado da mesma forma.

34

34

Grafos

- Antes de definirmos o tipo do grafo, vamos começar por definir o tipo das listas de adjacência.
- As listas de adjacência são implementadas com listas ligadas. Assim, vamos definir o tipo do nó dessas listas.
- Na implementação que se segue, consideramos que o tipo dos vértices do grafo é inteiro.
- O tipo da listas de adjacência é definido em C como um apontador para este nó da lista.

```
typedef int VERTICE;
typedef struct nodo {
        VERTICE v;
        struct nodo *next;
}NODO;

typedef NODO *ADJ;
typedef NODO *VERTICES;
```

35

35

Grafos

- O grafo pode ser definido como um array de apontadores para as listas de adjacência (array de inilistas).

  NODO *grafo[N]; // ou ADJ GRAFO[N];

- Em que N é o numero de vértices do grafo.
- O primeiro passo deverá ser a inicialização do array de listas ligadas, da seguinte forma:

```
void inicializaGrafo(NODO *grafo[]) {
        int i=0;
        for (i=0; i< MAX ; i++) {
                grafo[i]=NULL;
        }
}
```

36

36

Grafos

- Usando lista de adjacência facilmente se obtém os vértices sucessores de um dado vértice v. Os sucessores v são os vértices existentes na lista de adjacentes de v.

```
void showListaAdjacentes(ADJ g[],VERTICE v) {
        ADJ aux;
        if (g[v] == NULL)  {
                printf("Não existem adjacentes\n");
        }
        else {
                aux = g[v];
                while (aux!=NULL) {
                        printf("[ %d ]", aux->v);
                        aux=aux->seguinte;
                }
        }
}
```

37

37

Grafos

- Verificar se um dado vértice é sucessor de outro, corresponde a verificar se esse vértice está na respetiva lista de adjacentes.

```
int isSucessor(ADJ l,  VERTICE v) {
        if (l==NULL) return 0;
        if (l->v == v) return 1;
        return (isSucessor(l->seguinte,v));
}
```

Ao invocar a função poderá ser feita da seguinte forma:

```
....
if (isSucessor(grafo[orig], dest)) {
   printf("%d SIM é sucessor de %d\n", dest, orig);
}
else {
   printf("%d NAO é sucessor de %d\n", dest, orig);
}
```

38

38

Grafos

Para inserir um novo arco ao grafo com origem em v1 e destino v2 é necessário:

Em primeiro lugar reservar espaço de memória para o novo registo e em seguida acrescentar esse registo à lista de adjacência de v1.

```
int insLig(ADJ g[], VERTICE v1, VERTICE v2) {
NODO *novo, *aux;
novo=(NODO *)malloc(sizeof(NODO));
novo->v = v2;
novo->seguinte=NULL;
if(g[v1]==NULL)
        g[v1]= novo;
else {
        aux=g[v1];
        while (aux->seguinte != NULL) {
                aux=aux->seguinte;
        }
        aux->seguinte=novo;
}
return 0;
}
```

39

39

Grafos

Para verificar se existe caminho entre o vértice orig e o vértice dest é necessário:

- Verificar se o vértice destino é sucessor do vértice origem.
- Se assim for então existe caminho entre os dois vértices.
- Senão temos que verificar se algum dos sucessores do vértice origem tem caminho para o vértice destino.
- Para isso é necessário guardar os vértices já visitados (vis) para evitar revisitar vértices.

```
void haCaminho(ADJ g[], VERTICE orig, VERTICE dest){
        VERTICES vis=NULL,aux;
        if(haCaminhoAux(g,orig,dest,&vis)){
                printf("\nExiste caminho de %d para %d;", orig, dest);
                printf("\n%d ->", orig);
        }
        else {
           printf("Não existe caminho entre %d e %d\n", orig,dest);
        }
}
```

40

Grafos

- ...

```
int haCaminhoAux(ADJ g[], VERTICE orig, VERTICE dest, VERTICES *vis){
    ADJ suc=NULL;
    int b=0;
    marcaVisitado(&(*vis),orig);   // acrescentar a lista visitados
    if (isSucessor(g[orig],dest))  return 1;
    suc = g[orig];
    while((suc != NULL) && (!b)) {
            if (verificaVisitado(*vis,suc->v)==0 ) {
                printf("verificar entre %d e %d\n", suc->v,dest);
                b = haCaminhoAux(g,suc->v,dest,&(*vis));
             }
             suc = suc->seguinte;
    }
    return b;
}
```

41

41

Grafos

- A função marcaVisitados() vai acrescentando os vértices que vão sendo visitados à lista de vértices já visitados.

```
void marcaVisitado(NODO **l, VERTICE v1){
  NODO *novo, *aux;
  novo=(ADJ)malloc(sizeof(NODO));
  novo->v = v1;
  novo->seguinte=NULL;
  if ((*l)==NULL) {
          printf("Acrescenta o primeiro elemento\n");
          (*l)=novo;
  }
  else {
          aux=*l;
          printf("Acrescenta no fim %d\n", novo->v);
          while (aux->seguinte!=NULL)
                  aux=aux->seguinte;
          aux->seguinte=novo;
  }
}
```

42

42

Grafos

A função verificaVisitado(), tal como o nome indica, verifica se um determinado vértice, recebido por parâmetro, já foi visitado.

Para isso é necessário apenas percorrer a lista de vértices visitados e verificar se esse vértice pertence à lista.

```
int verificaVisitado(VERTICES l, VERTICE v) {
        VERTICES aux=l;
        if (l==NULL) return 0;
        while (aux!= NULL) {
                if (aux->v==v) return 1;
                aux=aux->seguinte;
        }
        return 0;
}
```

43

43

Grafos

Caminho mais curto ou de menor custo:

Existem vários algoritmo já estudados e conhecidos para descobrir o percurso, ou caminho mais curto entre dois pontos de um grafo. O algoritmo mais conhecido foi proposto por Edsger Dijkstra em 1959. Existem, no entanto, muito outros algoritmos propostos como o algoritmo de Bellman-Ford, o algoritmo de Floyd, etc.

O Algoritmo de Dijkstra:

- O algoritmo de Dijkstra, em cada etapa, escolhe a melhor solução possível considerando que a escolha das sucessivas melhores escolhas produz a melhor solução final.
- Este algoritmo não encontra só o caminho mais curto entre dois vértices, mas encontra o caminho mais curto entre um vértice (chamada origem) e todas os outros vértices do grafo.

44

44

Grafos

Os passos do **algoritmo de Dijkstra** são os seguinte:

De inicio (inicialização):

- **Todos os vértices são considerados desconhecidos** (ou não visitados);
- **Todos os caminhos têm um peso maior** que qualquer um dos peso reais;
- O vértice predecessor a cada vértice não é conhecido;
- O vértice origem tem um caminho com custo 0 e não tem predecessor.

Em seguida:

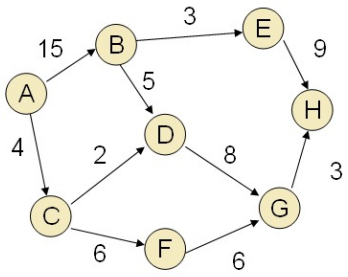
- Em cada etapa do algoritmo **é escolhido o vértice cujo caminho tem menor custo**, começando com o vértice origem, que é marcado como visitado;
- Para todos os adjacentes ainda não visitados é **calculado o custo da travessia desses vértices**. Se o custo calculado for menor que o custo anteriormente atribuído ao vértice, então este caminho alternativo é melhor do que o anterior.
- Em seguida são **atualizados os custo e os predecessores**.
- Enquanto existirem vértices não visitados, o algoritmo **escolhe o vértice que entre eles tem menor custo.**

45

45

Grafos

Exemplo 1: Para o grafo seguinte, representar os passos seguidos pelo **algoritmo de Dijkstra**, para o nó origem A:

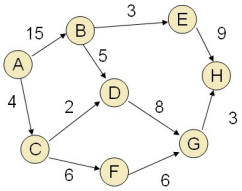


- Inicializar: Colocar todos os nós com custo infinito com exceção do vértice origem (A) que fica com custo 0 (zero). (1ª linha tabela seguinte)
- Explorar o vértice A: marcar como visitado e verificar os seus adjacentes. Temos um arco com peso 15 para B e um arco com peso 4 para C. O vértice B fica com custo de A (0) + 15 porque é menor que infinito. O vértice C fica com custo de A (0) + 4, uma vez que é menor que infinito. (2ª linha tabela seguinte)
- Selecionar o vértice adjacente a A, ainda não visitado, com menor custo: vértice C.

46

46

Grafos



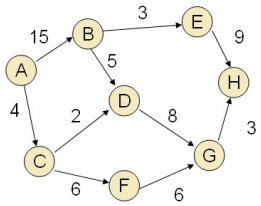
- Explorar C: marcar como visitado e verificar os seus adjacentes: D e F. D fica com custo de C (4) + 2 e F fica com custo de C (4)+6 porque este custo é inferior ao anterior (infinito). (linha 3 da tabela)
- Selecionar adjacente a C, ainda não visitado, com menor custo: vértice D.
- Explorar D: …
- ……
- Até todos os vértices estarem visitados.

Orig	A	B	C	D	E	F	G	H
Ini	**0 A**	INF	INF	INF	INF	INF	INF	INF
A	0 A	15 A	**4 A**	INF	INF	INF	INF	INF
C	0 A	15 A	4 A	**6 C**	INF	10 C	INF	INF
D	0 A	15 A	4 A	6 C	INF	**10 C**	14 D	INF
F	0 A	15 A	4 A	6 C	INF	10 C	**14 D**	INF
G	0 A	**15 A**	4 A	6 C	INF	10 C	14 D	17 G
B	0 A	15 A	4 A	6 C	18 B	10 C	14 D	**17 G**
H	0 A	15 A	4 A	6 C	**18 B**	10 C	14 D	17 G
E	0 A	15 A	4 **A**	6 **C**	18 B	10 C	14 **D**	17 **G**

47

47

Grafos



O caminho mais curto de A para:

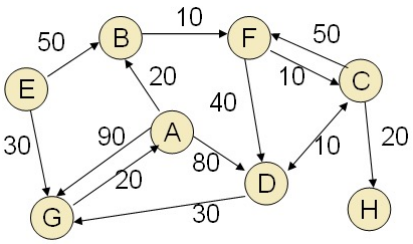
- H: tem custo 17 e é = H-G-D-C-A
- G: tem custo 14 e é = G-D-C-A
- F: tem custo 10 e é = F-C-A
- E: tem custo 18 e é: E-B-A
- D: tem custo 6 e é: D-C-A
- C: tem custo 4 e é: C-A
- B: tem custo 15 e é: B-A

Orig	A	B	C	D	E	F	G	H
Ini	**0 A**	INF	INF	INF	INF	INF	INF	INF
A	0 A	15 A	**4 A**	INF	INF	INF	INF	INF
C	0 A	15 A	4 A	**6 C**	INF	10 C	INF	INF
D	0 A	15 A	4 A	6 C	INF	**10 C**	14 D	INF
F	0 A	15 A	4 A	6 C	INF	10 C	**14 D**	INF
G	0 A	**15 A**	4 A	6 C	INF	10 C	14 D	17 G
B	0 A	15 A	4 A	6 C	18 B	10 C	14 D	**17 G**
H	0 A	15 A	4 A	6 C	**18 B**	10 C	14 D	17 G
E	0 A	15 A	4 **A**	6 **C**	18 B	10 C	14 **D**	17 **G**

48

48

Grafos (Exemplo2 algoritmo de Dijkstra)



Algumas conclusões:

- O caminho mais perto entre A e H tem custo 60 e é: H-C-F-B-A.
- O caminho mais curto entre A e G tem custo 80 e é: G-D-C-F-B-A
- Não existe caminho de A para E.
- etc.

	A	B	C	D	E	F	G	H
Ini	**0 A**	IN F	INF	INF	INF	INF	INF	INF
A	0 A	**20 A**	INF	**80 A**	INF	INF	**90 A**	INF
B	0 A	20 A	INF	80 A	INF	**30 B**	90 A	INF
F	0 A	20 A	**40 F**	**70 F**	INF	30 B	90 A	INF
C	0 A	20 A	40 F	**50 C**	INF	30 B	90 A	**60 C**
D	0 A	20 A	40 F	50 C	INF	30 B	**80 D**	**60 C**
H	0 A	20 A	40 F	50 C	INF	30 B	**80 D**	60 C
G	0 A	20 A	40 F	50 C	INF	30 B	80 D	60 C

49

49

Grafos

O algoritmo de Dijkstra escrito em pseudocódigo:

```
Dijkstra (grafo, origem, predecessor[], custo[], visitado[])
para (todos os nós i do grafo)
        custo(i) = infinito             // inicializar com um nº suficientemente grande
        visitado(i) = Falso
        predecessor(i) = -1             // inicializar com -1 - não há caminho até ao nó
fim para
Custo (origem) = 0                      // inicializar origem com custo 0
enquanto (nos_visitados < tamanho_do_grafo)  // enquanto existem vértices por visitar
        selMenorCusto (custo(i));       // selecionar o vértice com menor custo
        visitado(i) = Verdadeiro        // colocamos o vértice i como visitado

                        // vamos atualizar as distância para todos os vizinhos de i
        Para (todos os vizinhos j do vértice i)
                se (custo(i) + peso(i,j) < custo(j)) então
                        custo(j) = custo(i) + peso(i,j)
                        predecessor(j) = i
                fim se
        fim para
Fim enquanto
```

50

50

Bibliografia

- Programação Avançada Usando C, António Manuel Adrego da Rocha, ISBN: 978-978-722-546-0.
- Schildt, Herbert: C the complete Reference, McGraw-Hill, 1998.
- Algoritmia e Estruturas de Dados, José Braga de Vasconcelos, João Vidal de Carvalho, ISBN: 989-615-012-5.
- Elementos de Programação com C - Pedro João Valente D. Guerreiro, 3ª edição, ISBN: 972-722-510-1.
- Introdução à Programação Usando C, António Manuel Adrego da Rocha, ISBN: 972-722-524-1.