MC 202 EF - 2s2007

Grafos

prof. Fernando Vanini
IC-UNICAMP
Klais Soluções

Grafos

Definição:

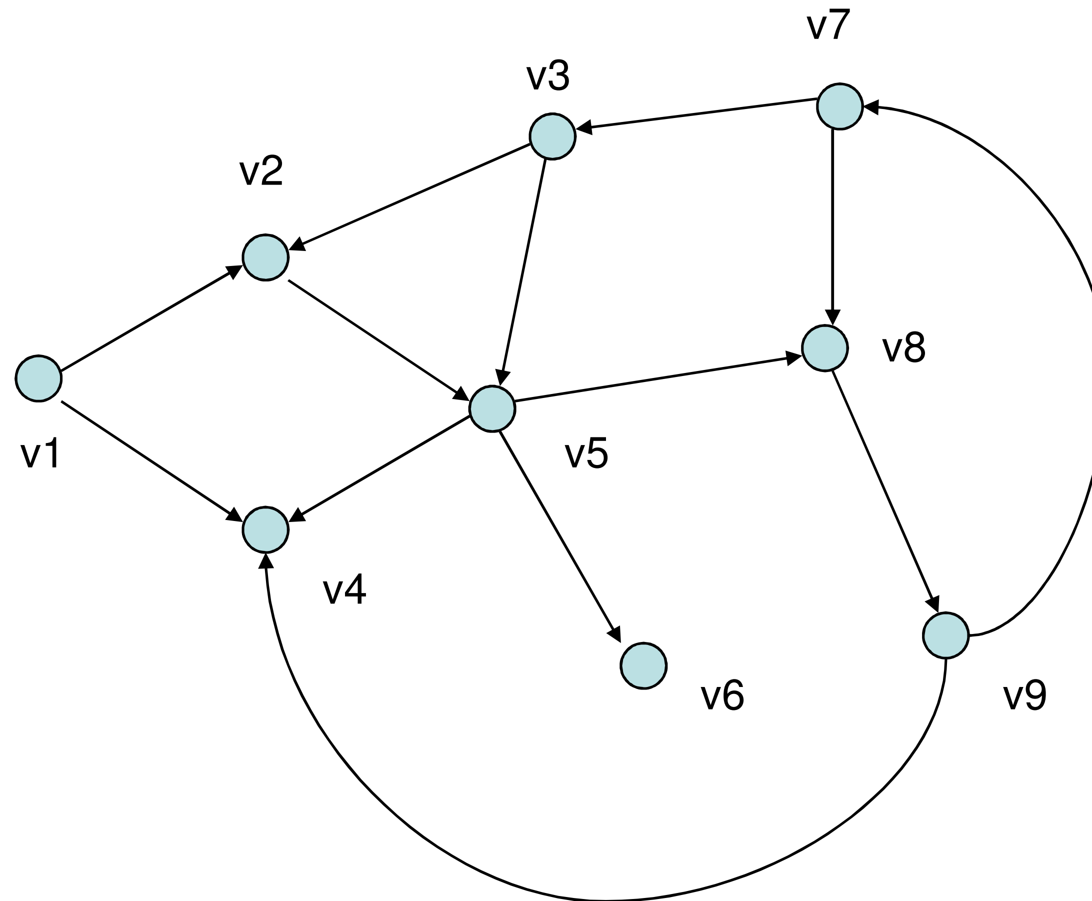
- um grafo dirigido G é formado por
  - um conjunto finito V de vértices
  - um conjunto de arestas $A \subseteq V \times V$
- uma aresta *liga* ou *conecta* dois vértices.
- um grafo não dirigido é um grafo no qual as arestas são pares não ordenados.

Grafos

Aplicações: grafos são muito utilizados para modelar sistemas reais como por exemplo:

- redes de distribuição de energia, telecomunicações
- malha viária de uma cidade
- circuitos elétricos
- processos industriais e processos de negócio

Grafos - um exemplo



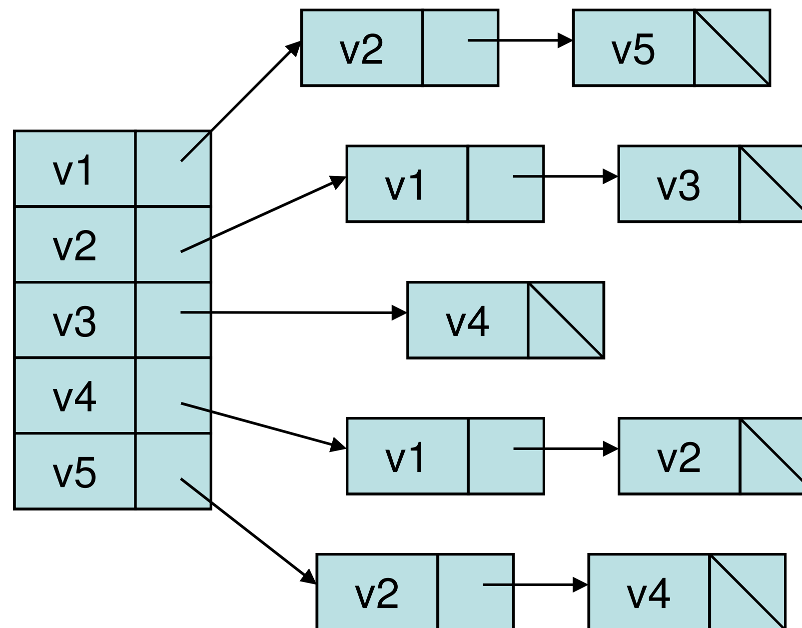
Grafos - representação

- Matriz de adjacência:
  – o grafo é representado por uma matriz quadrada M onde o elemento M[i,j] é 1 ou 0 indicando se existe ou não uma aresta ligando Vi a Vj.

	V1	V2	V3	V4	V5
V1	0	1	0	0	1
V2	1	0	1	0	0
V3	0	0	0	1	0
V4	1	1	0	0	0
V5	0	1	0	1	0

Grafos - representação

- Listas de adjacência:
  - cada vértice Vi do grafo é representado por uma lista que contém os vértices Vj para os quais existe a aresta (Vi,Vj)



Grafos - percursos padrão

Percurso *em largura* a partir de um vértice Vi:

```
percurso_em_largura(vértice Vi){
    fila F = fila vazia; vértice V;
    inserir Vi em F;
    enquanto F não for vazia {
        retirar V da fila F;
        se V ainda não foi visitado {
                visita V; marca V como 'visitado';
        }
        para todo vértice W adjacente a V {
                inserir W na fila F;
        }
    }
}
```

Grafos - percursos padrão

Percurso *em profundidade* a partir de um vértice Vi:

```
percurso_em_profundidade(vértice Vi){
    pilha P = pilha vazia; vértice V;
    inserir Vi em P;
    enquanto P não for vazia {
        desempilhar V;
        se V ainda não foi visitado {
            visita V; marca V como 'visitado';
        }
        para todo vértice W adjacente a V  { empilhar W ;  }
    }
}
```

Grafos - representação em C

Arestas:

```
/* representação de uma aresta */
typedef struct edge* apEdge;
typedef struct edge {
  int    c;    /* 'peso' ou 'custo' associado à aresta */
  int    v;    /* índice do vértice 'destino'          */
  apEdge next; /* próxima aresta da lista              */
} edge;
```

Grafos - representação em C

Vértices:

```
typedef struct vert* apVert;
typedef struct edge* apEdge;

/* descrição de um vértice */
typedef struct vert {
  int    d;    /* 'distância' do vértice à 'origem'       */
  int    r;    /* próximo vértice no caminho até a origem */
  int    m;    /* vértice marcado ?                       */
  apEdge list; /* lista de arestas                        */
} vert;
```

Grafo:

```
struct vert graph[n];
```

Grafos - representação em C

Percurso em profundidade (versão recursiva):

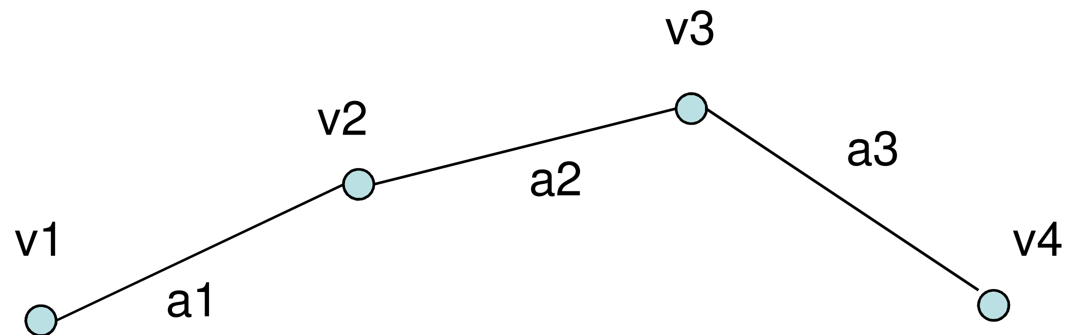
```
void depthFirst(apVert g, int v){
  apEdge e;
  if(!g[v].m){
    printf(" %d,",v);
    g[v].m = true;
    e = g[v].list;
    while(e != NULL){
      depthFirst(g,e->v);
      e = e->next;
    }
  }
}
```

Grafos - percurso em largura

```
void breathFirst(apVert g, int v){
   int w; apEdge e;
   qInit();
   qInsert(v); g[v].m = true;
   while((w=qRemove())!= NULLNODE){
     printf("w:%d ",w);
     e = g[w].list;
     while(e != NULL){
       if(!g[e->v].m) {
         g[e->v].m = true;
         qInsert(e->v);
       }
       e = e->next;
     }
   }
}
```

Caminho num grafo

- Um caminho num grafo G = (V,E) é uma seqüência de arestas a_1, a_2 ... $a_{k,,}$ pertencentes a E tais que
    - a1 = (v_1, v_2)
    - a2 = (v_2, v_3)
    - ...
    - $a_{k-1} = (v_{k-1},v_k)$
    - $a_k = (v_k, v_{k+1})$

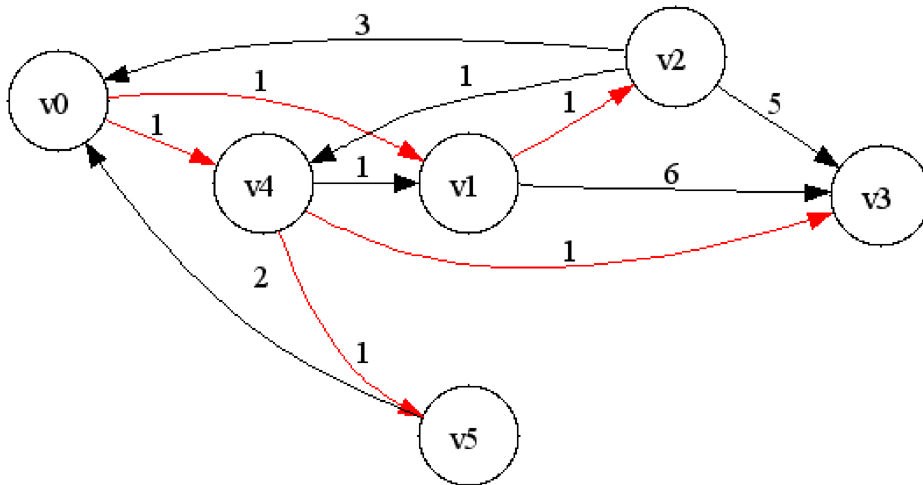


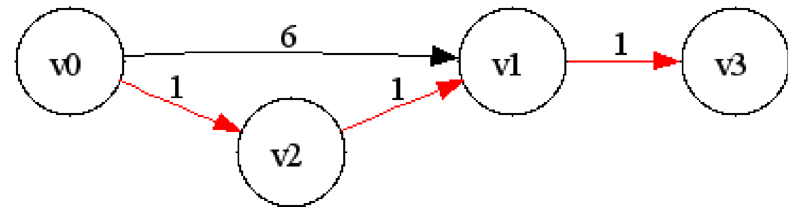
Caminho mínimo

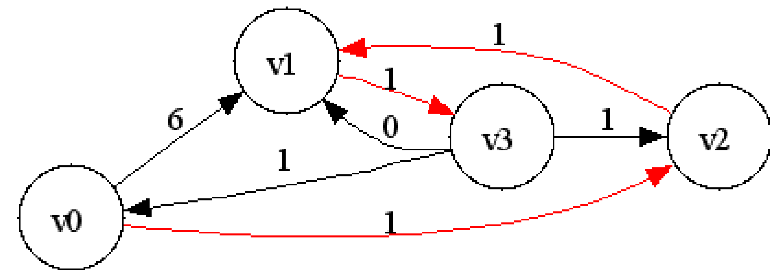
- Dado um grafo G = (V,E), um caminho C, de um vértice v a um vértice w, é mínimo em G se ele for o caminho de v a w em que a soma das 'distâncias' em cada uma das suas arestas for mínima.

Exemplos

- Nos exemplos a seguir, as arestas em vermelho fazem parte da *árvore dos caminhos mínimos* a partir da origem (v0).







Algoritmo de Dijkstra

- O algoritmo de Dijkstra, usa uma abordagem parecida com o percurso em largura, utilizando a ‘proximidade’ de cada nó à origem como critério para inclui-lo na seqüência de visitas.

Algoritmo de Dijkstra

```
caminhos_mínimos (grafo g, vértice v) {
    iniciar o grafo;  iniciar a fila de prioridades pQ;
    enquanto  pQ não vazia {
        remover w de pQ; /* dist(v,w) é mínima */
        para todo x adjacente a w {
            nd = dist(v,w)+dist(w,x);
            se (nd < dist(v,x)) {
                dist(v,x) = nd;
                rearranjar pQ;  /* dist(v,x) foi alterada */
            }
        }
    }
}
```

Algoritmo de Dijkstra

```
iniciar o grafo( ){
    para todo vértice w {
        se E contém a aresta (v,w)
            então dist(v,w) = distância 'direta' de v a w;
            senão dist(v,w) = ∞ ;
    }
    iniciar fila de prioridades pQ( ) {
        para todo vértice w
            inserir w em pQ de acordo com  dist(v,w);
    }
```

Algoritmo de Dijkstra

- Tempo de execução
  - Cada aresta do grafo é visitada uma vez e a cada visita pode ocorrer um rearranjo da fila de prioridades.
  - Se a fila de prioridades for implementada como um heap, cada rearranjo vai levar um tempo proporcional a $log_2 |V|$.
  - O tempo de execução será portanto proporcional a $|E| log_2 |V|$ ou $|V|^2 log_2 |V|$.

Caminho mínimo p/ todos os pares de vértices

Algoritmo de Floyd-Warshall

- Grafo representado como uma matriz M onde M[i][j] representa o 'custo' da aresta (i,j):
  - Se (i,j) Є E , M[i][j] = custo(i,j) senão M[i][j] = ∞.

```
for(k = 0; k < N; k++)
  for(i = 0; i < N; i++)
     for(j = 0; j< N; j++){
       int s = M[i][k] + M[k][j];
       if( s < M[i][j])
          M[i][j] = s;
     }
```

- O tempo de execução é proporcional a N^3.

Caminho mínimo p/ todos os pares de vértices

Identificando os caminhos

- O algoritmo apresentado só determina o custo do menor caminho para cada par de vértices (i,j).
- Para determinar os caminhos, pode-se usar uma matriz R onde cada elemento (i,j) define o próximo trecho do 'caminho reverso' de i a j (inicialmente, R[i][j] = NULL p/ todo par (i,j)).

```
for(k = 0; k < N; k++)
  for(i = 0; i < N; i++)
     for(j = 0; j < N; j++){
       int s = M[i][k] + M[k][j];
       if( s < M[i][j])
          M[i][j] = s; R[i][j] = k;
     }
```

Caminho mínimo p/ todos os pares de vértices

Identificando os caminhos

- Tendo executado o algoritmo, para se obter o caminho mínimo de um vértice i até um vértice j temos que refazer cada trecho do caminho a partir de R[i][j], o que é feito na função abaixo.

```
void printPath(int i, int j){
  if(R[i][j] == NULL) printf("aresta %d - %d\n",i,j);
  else {
     printPath(i,R[i][j]);
     printPath(R[i][j],j);
  }
}
```

Árvore geradora de um grafo

- Dado G = (V,E) onde V é o conjunto de vértices e E é o conjunto de arestas, o grafo G' = (V,A), onde A $\subseteq$ E é uma árvore geradora de G se
  - G' não contém ciclos e
  - Se em G existir um caminho de v para w então em G' também existe um caminho de v para w ou de w para v.

Construção da árvore geradora

Entrada: grafo G = (V,E)

<u>início</u>

A = conjunto vazio de arestas;

<u>para toda</u> aresta a de E <u>faça</u>

<u>se</u> a não forma ciclo com as arestas em A

<u>então</u> insira a em A;

<u>fim;</u>

"a forma ciclo com as arestas em A" ?

- ao longo do processo de construção da árvore, são montados *pedaços* da mesma que vão se juntando à medida em que as arestas são inseridas .
- uma aresta a = (v1, v2) vai formar ciclo com as arestas em A se v1 e v2 estiverem no mesmo *pedaço* da árvore.
- uma forma eficiente de verificar se uma aresta a forma ciclo com as arestas em A é
  - considerar cada *pedaço* da árvore como uma classe de equivalência
  - a = (v1, v2) forma ciclo com as arestas em A se v1 e v2 estiverem na mesma classe de equivalência.
  - quando uma aresta a = (v1, v2) for inserida na árvore, as classes de equivalência de v1 e v2 passam a ser uma única classe (os *pedaços* se juntam).

Classes de equivalência

- Representação
  - cada vértice mantém um apontador para o 'representante' da sua classe. Para o representante, esse apontador é igual a NULL.
  - no início do algoritmo, quando a árvore é vazia, cada vértice é o representante da sua própria classe.

```
/* descrição de um vértice */
typedef struct vert {
  int     d;   /* 'distância' do vértice à 'origem'           */
  int     r;   /* próximo vértice no caminho até a origem     */
  int     m;   /* vértice marcado ?                           */
  apVert  s;   /* aponta para o 'representante' deste vértice */
  apEdge list; /* lista de arestas iniciando neste vértice    */
} vert;
```

Classes de equivalência

Operações

- determinar a classe de equivalência de um vértice v

```
/* devolve a 'raiz' da classe de equiv. de um vértice */
apVert super(apVert v){
  if(v->s == NULL) return v;
  return (v->s = super(v->s));
}
```

A cada busca pela classe de equivalência, os vértices intermediários também são atualizados, de forma que as próximas buscas serão mais rápidas.

Classes de equivalência

Operações

- juntar as classes de equivalência dos vértices v e w

```
/* coloca v e w na mesma classe de equivalência */
void makeEquiv(apVert v, apVert w){
  apVert sv = super(v);
  apVert sw = super(w);
  if(sv != sw) sv->s = sw;
}
```

Construção da árvore geradora

As arestas da árvore podem ser mantidas numa lista onde cada nó é do tipo:

```
typedef struct spTreeEdge* apSpTreeEdge;
typedef struct spTreeEdge {
  int      v1;          /* primeiro vértice  */
  int      v2;          /* segundo vértice   */
  apSpTreeEdge next; /* próximo da lista  */
} spTreeEdge;
```

Construção da árvore geradora

```
/* árvore geradora de um grafo g com n vértices */
apSpTreeEdge makeSpanningTree(apVert g, int n){
   int i;
   apSpTreeEdge spTree = NULL;
   for(i = 0; i < n; i++){
     apEdge e = g[i].list;
     while(e != NULL){
       if(super(&g[i]) != super(&g[e->v])){
         makeEquiv(&g[i],&g[e->v]);
         spTree = newSpEdge(i,e->v,spTree);
       }
       e = e->next;
     }
   }
   return spTree;
}
```

MC 202 EF - 2s2007

Grafos

prof. Fernando Vanini
IC-UNICAMP
Klais Soluções

Grafos

Definição:

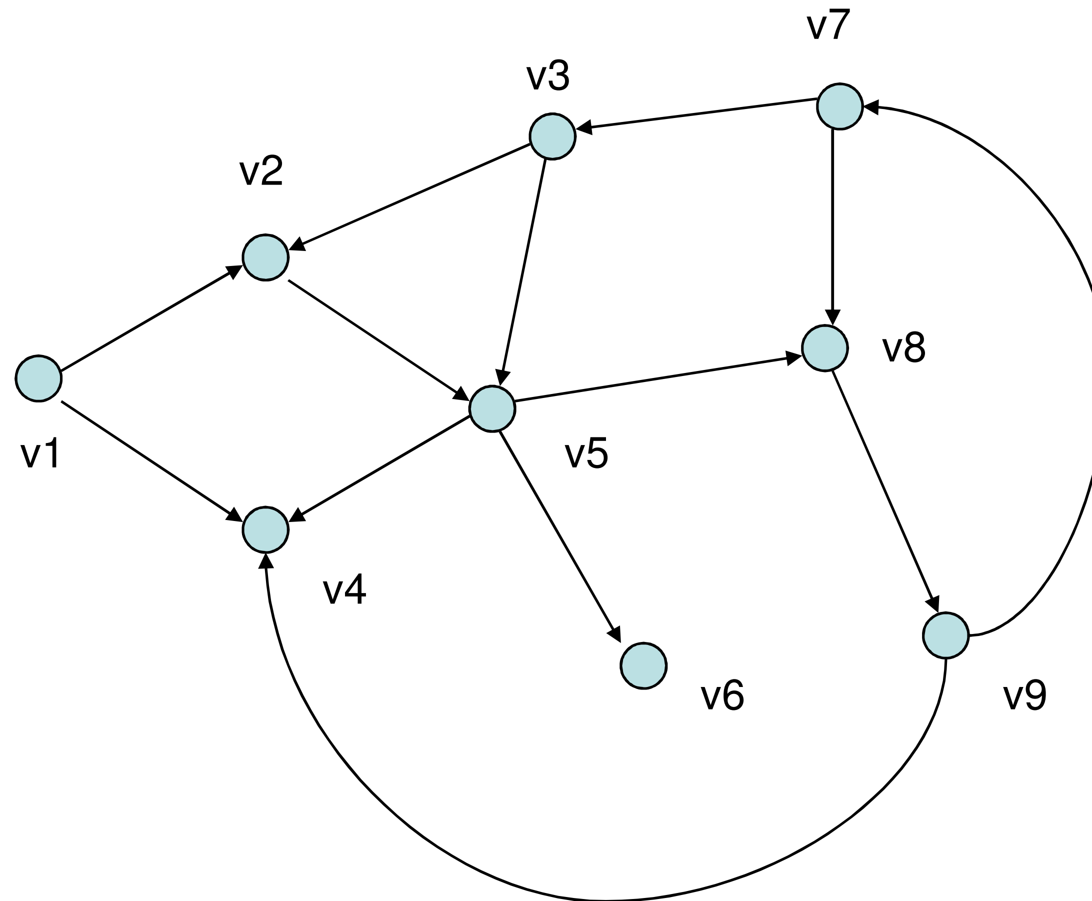
- um <u>grafo dirigido</u> G é formado por
  - um conjunto finito V de vértices
  - um conjunto de arestas $A \subseteq V \times V$
- uma aresta *liga* ou *conecta* dois vértices.
- um grafo não dirigido é um grafo no qual as arestas são pares não ordenados.

Grafos

Aplicações: grafos são muito utilizados para modelar sistemas reais como por exemplo:

- redes de distribuição de energia, telecomunicações
- malha viária de uma cidade
- circuitos elétricos
- processos industriais e processos de negócio

Grafos - um exemplo



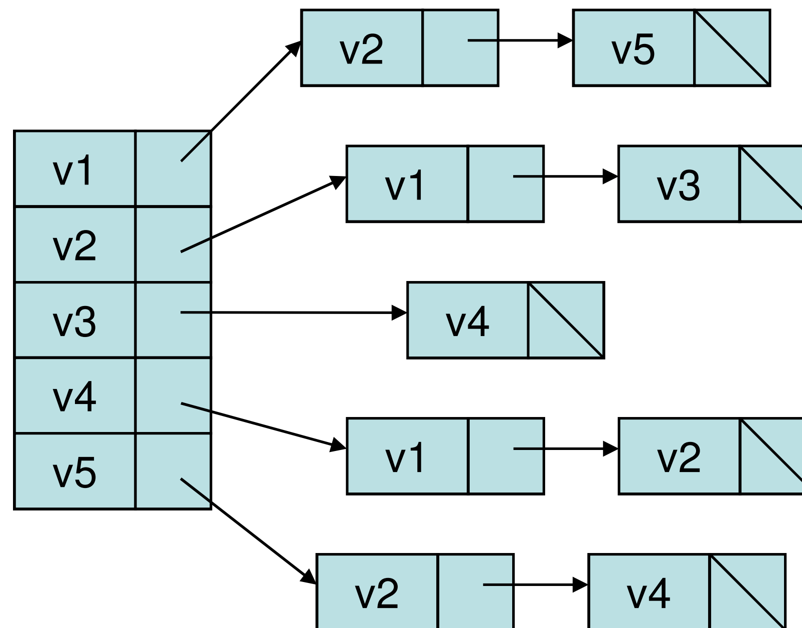
Grafos - representação

- Matriz de adjacência:
  - o grafo é representado por uma matriz quadrada M onde o elemento M[i,j] é 1 ou 0 indicando se existe ou não uma aresta ligando Vi a Vj.

	V1	V2	V3	V4	V5
V1	0	1	0	0	1
V2	1	0	1	0	0
V3	0	0	0	1	0
V4	1	1	0	0	0
V5	0	1	0	1	0

Grafos - representação

- Listas de adjacência:
  - cada vértice Vi do grafo é representado por uma lista que contém os vértices Vj para os quais existe a aresta (Vi,Vj)



Grafos - percursos padrão

Percurso *em largura* a partir de um vértice Vi:

```
percurso_em_largura(vértice Vi){
    fila F = fila vazia; vértice V;
    inserir Vi em F;
    enquanto F não for vazia {
        retirar V da fila F;
        se V ainda não foi visitado {
                visita V; marca V como 'visitado';
        }
        para todo vértice W adjacente a V {
                inserir W na fila F;
        }
    }
}
```

Grafos - percursos padrão

Percurso *em profundidade* a partir de um vértice Vi:

```
percurso_em_profundidade(vértice Vi){
    pilha P = pilha vazia; vértice V;
    inserir Vi em P;
    enquanto P não for vazia {
        desempilhar V;
        se V ainda não foi visitado {
            visita V; marca V como 'visitado';
        }
        para todo vértice W adjacente a V { empilhar W ; }
    }
}
```

Grafos - representação em C

Arestas:

```
/* representação de uma aresta */
typedef struct edge* apEdge;
typedef struct edge {
  int    c;    /* 'peso' ou 'custo' associado à aresta */
  int    v;    /* índice do vértice 'destino'          */
  apEdge next; /* próxima aresta da lista              */
} edge;
```

Grafos - representação em C

Vértices:

```
typedef struct vert* apVert;
typedef struct edge* apEdge;

/* descrição de um vértice */
typedef struct vert {
  int    d;    /* 'distância' do vértice à 'origem'       */
  int    r;    /* próximo vértice no caminho até a origem */
  int    m;    /* vértice marcado ?                       */
  apEdge list; /* lista de arestas                        */
} vert;
```

Grafo:

```
struct vert graph[n];
```

Grafos - representação em C

Percurso em profundidade (versão recursiva):

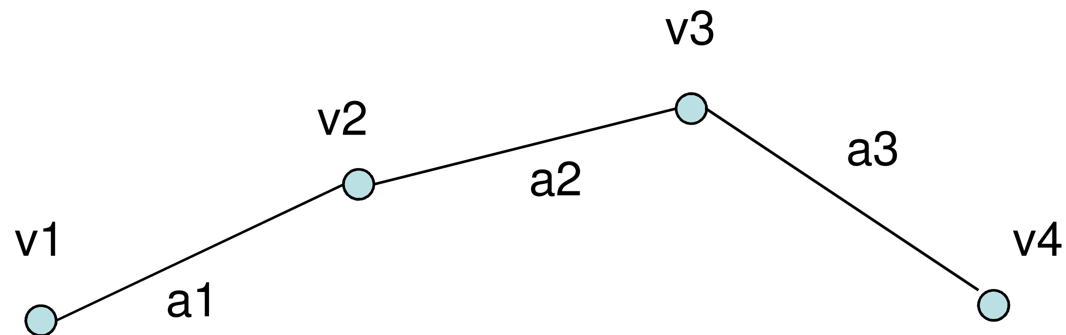
```
void depthFirst(apVert g, int v){
  apEdge e;
  if(!g[v].m){
    printf(" %d,",v);
    g[v].m = true;
    e = g[v].list;
    while(e != NULL){
      depthFirst(g,e->v);
      e = e->next;
    }
  }
}
```

Grafos - percurso em largura

```
void breathFirst(apVert g, int v){
   int w; apEdge e;
   qInit();
   qInsert(v); g[v].m = true;
   while((w=qRemove())!= NULLNODE){
     printf("w:%d ",w);
     e = g[w].list;
     while(e != NULL){
       if(!g[e->v].m) {
         g[e->v].m = true;
         qInsert(e->v);
       }
       e = e->next;
     }
   }
}
```

Caminho num grafo

- Um caminho num grafo G = (V,E) é uma seqüência de arestas a_1, a_2 ... $a_{k,,}$ pertencentes a E tais que
  - a1 = (v_1, v_2)
  - a2 = (v_2, v_3)
  - ...
  - a_{k-1} = (v_{k-1},v_k)
  - a_k = (v_k, v_{k+1})

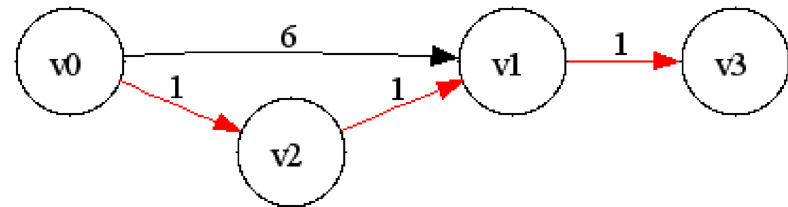


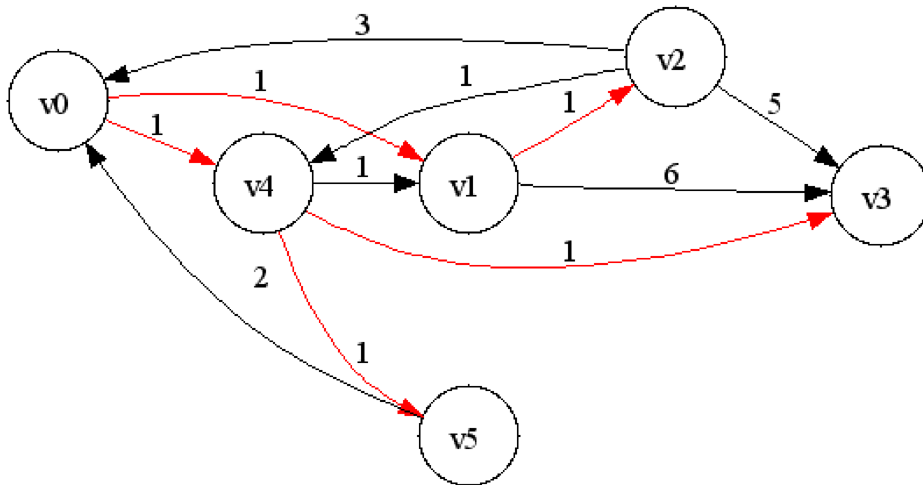
Caminho mínimo

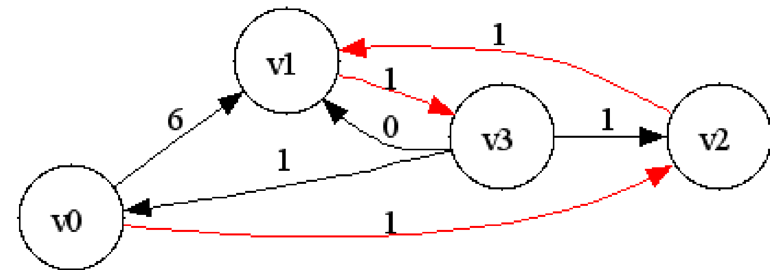
- Dado um grafo G = (V,E), um caminho C, de um vértice v a um vértice w, é mínimo em G se ele for o caminho de v a w em que a soma das 'distâncias' em cada uma das suas arestas for mínima.

Exemplos

- Nos exemplos a seguir, as arestas em vermelho fazem parte da *árvore dos caminhos mínimos* a partir da origem (v0).







Algoritmo de Dijkstra

- O algoritmo de Dijkstra, usa uma abordagem parecida com o percurso em largura, utilizando a ‘proximidade’ de cada nó à origem como critério para inclui-lo na seqüência de visitas.

Algoritmo de Dijkstra

```
caminhos_mínimos (grafo g, vértice v) {
    iniciar o grafo;  iniciar a fila de prioridades pQ;
    enquanto  pQ não vazia {
        remover w de pQ; /* dist(v,w) é mínima */
        para todo x adjacente a w {
            nd = dist(v,w)+dist(w,x);
            se (nd < dist(v,x) {
                dist(v,x) = nd;
                rearranjar pQ;  /* dist(v,x) foi alterada */
            }
        }
    }
}
```

Algoritmo de Dijkstra

```
iniciar o grafo( ){
    para todo vértice w {
        se E contém a aresta (v,w)
            então dist(v,w) = distância 'direta' de v a w;
            senão dist(v,w) = ∞ ;
    }
    iniciar fila de prioridades pQ( ) {
        para todo vértice w
            inserir w em pQ de acordo com  dist(v,w);
    }
```

Algoritmo de Dijkstra

- Tempo de execução
  - Cada aresta do grafo é visitada uma vez e a cada visita pode ocorrer um rearranjo da fila de prioridades.
  - Se a fila de prioridades for implementada como um heap, cada rearranjo vai levar um tempo proporcional a $log_2 |V|$.
  - O tempo de execução será portanto proporcional a $|E| log_2 |V|$ ou $|V|^2 log_2 |V|$.

Caminho mínimo p/ todos os pares de vértices

Algoritmo de Floyd-Warshall

- Grafo representado como uma matriz M onde M[i][j] representa o 'custo' da aresta (i,j):
  - Se (i,j) Є E , M[i][j] = custo(i,j) senão M[i][j] = ∞.

```
for(k = 0; k < N; k++)
  for(i = 0; i < N; i++)
     for(j = 0; j< N; j++){
       int s = M[i][k] + M[k][j];
       if( s < M[i][j])
          M[i][j] = s;
     }
```

- O tempo de execução é proporcional a N^3.

Caminho mínimo p/ todos os pares de vértices

Identificando os caminhos

- O algoritmo apresentado só determina o custo do menor caminho para cada par de vértices (i,j).
- Para determinar os caminhos, pode-se usar uma matriz R onde cada elemento (i,j) define o próximo trecho do 'caminho reverso' de i a j (inicialmente, R[i][j] = NULL p/ todo par (i,j)).

```
for(k = 0; k < N; k++)
  for(i = 0; i < N; i++)
     for(j = 0; j < N; j++){
       int s = M[i][k] + M[k][j];
       if( s < M[i][j])
          M[i][j] = s; R[i][j] = k;
     }
```

Caminho mínimo p/ todos os pares de vértices

Identificando os caminhos

- Tendo executado o algoritmo, para se obter o caminho mínimo de um vértice i até um vértice j temos que refazer cada trecho do caminho a partir de R[i][j], o que é feito na função abaixo.

```
void printPath(int i, int j){
  if(R[i][j] == NULL) printf("aresta %d - %d\n",i,j);
  else {
     printPath(i,R[i][j]);
     printPath(R[i][j],j);
  }
}
```

Árvore geradora de um grafo

- Dado $G = (V,E)$ onde V é o conjunto de vértices e E é o conjunto de arestas, o grafo $G' = (V,A)$, onde $A \subseteq E$ é uma árvore geradora de G se
  - G' não contém ciclos e
  - Se em G existir um caminho de v para w então em G' também existe um caminho de v para w ou de w para v.

Construção da árvore geradora

Entrada: grafo G = (V,E)

início

A = conjunto vazio de arestas;

para toda** aresta a de E **faça

***se** a não forma ciclo com as arestas em A*

***então** insira a em A;*

fim;

"a forma ciclo com as arestas em A" ?

- ao longo do processo de construção da árvore, são montados *pedaços* da mesma que vão se juntando à medida em que as arestas são inseridas .
- uma aresta a = (v1, v2) vai formar ciclo com as arestas em A se v1 e v2 estiverem no mesmo *pedaço* da árvore.
- uma forma eficiente de verificar se uma aresta a forma ciclo com as arestas em A é
  – considerar cada *pedaço* da árvore como uma classe de equivalência
  – a = (v1, v2) forma ciclo com as arestas em A se v1 e v2 estiverem na mesma classe de equivalência.
  – quando uma aresta a = (v1, v2) for inserida na árvore, as classes de equivalência de v1 e v2 passam a ser uma única classe (os *pedaços* se juntam).

Classes de equivalência

- Representação
  - cada vértice mantém um apontador para o 'representante' da sua classe. Para o representante, esse apontador é igual a NULL.
  - no início do algoritmo, quando a árvore é vazia, cada vértice é o representante da sua própria classe.

```
/* descrição de um vértice */
typedef struct vert {
  int     d;   /* 'distância' do vértice à 'origem'            */
  int     r;   /* próximo vértice no caminho até a origem      */
  int     m;   /* vértice marcado ?                            */
  apVert  s;   /* aponta para o 'representante' deste vértice */
  apEdge list; /* lista de arestas iniciando neste vértice     */
} vert;
```

Classes de equivalência

Operações

- determinar a classe de equivalência de um vértice v

```
/* devolve a 'raiz' da classe de equiv. de um vértice */
apVert super(apVert v){
  if(v->s == NULL) return v;
  return (v->s = super(v->s));
}
```

A cada busca pela classe de equivalência, os vértices intermediários também são atualizados, de forma que as próximas buscas serão mais rápidas.

Classes de equivalência

Operações

- juntar as classes de equivalência dos vértices v e w

```
/* coloca v e w na mesma classe de equivalência */
void makeEquiv(apVert v, apVert w){
  apVert sv = super(v);
  apVert sw = super(w);
  if(sv != sw) sv->s = sw;
}
```

Construção da árvore geradora

As arestas da árvore podem ser mantidas numa lista onde cada nó é do tipo:

```
typedef struct spTreeEdge* apSpTreeEdge;
typedef struct spTreeEdge {
  int      v1;         /* primeiro vértice  */
  int      v2;         /* segundo vértice   */
  apSpTreeEdge next; /* próximo da lista  */
} spTreeEdge;
```

Construção da árvore geradora

```
/* árvore geradora de um grafo g com n vértices */
apSpTreeEdge makeSpanningTree(apVert g, int n){
   int i;
   apSpTreeEdge spTree = NULL;
   for(i = 0; i < n; i++){
     apEdge e = g[i].list;
     while(e != NULL){
       if(super(&g[i]) != super(&g[e->v])){
         makeEquiv(&g[i],&g[e->v]);
         spTree = newSpEdge(i,e->v,spTree);
       }
       e = e->next;
     }
   }
   return spTree;
}
```