```
/*TAD FILA*/
typedef struct
{
   int indMemoria;
   int indInf;
}DADOS;
typedef struct nodo
{
   DADOS inf;
   struct nodo * next;
}NODO;
typedef struct
{
   NODO *INICIO;
   NODO *FIM;
}DESCRITOR;
typedef DESCRITOR * FILA_ENC;
void cria_fila (FILA_ENC *);
int eh_vazia (FILA_ENC);
void ins (FILA_ENC, DADOS);
DADOS cons (FILA_ENC);
void ret (FILA_ENC);
DADOS cons_ret (FILA_ENC);
```

```
void buscaEmLargura(listaDeNodos node, int G, int s)
{
  int u, *d=NULL, *pai=NULL, *vertice=NULL, numVertices=0, v, ind;
  char *cor=NULL; /*'B'=Branco, 'C'=Cinza e 'P'=Preto*/
  FILA_ENC Q;
  DADOS aux;
  u = G;
  while (u>=0)
  {
    ++numVertices;
    d = (int *) realloc(d, numVertices*sizeof(int));
    pai = (int *) realloc(pai, numVertices*sizeof(int));
    cor = (char *) realloc(cor, numVertices*sizeof(char));
    vertice = (int *) realloc(vertice, numVertices*sizeof(int));
/*apesar de suprimida é necessária a verificação das alocações dinâmicas*/
    if (u != s)
    {
      cor[numVertices-1] = 'B';
      d[numVertices-1] = -1; /*-1 equivale a infinito*/
      pai[numVertices-1] = -1; /*-1 equivale a NULL*/
      vertice[numVertices-1] = node[u].info;
    }
```

```
    else
    {
      ind = numVertices-1;
      cor[numVertices-1] = 'C';
      d[numVertices-1] = 0;
      pai[numVertices-1] = -1; /*-1 equivale a NULL*/
      vertice[numVertices-1] = node[u].info;
    }
    u = node[u].next;
  }
  cria_fila(&Q);
  aux.indInf = s;
  aux.indMemoria = ind;
  ins(Q, aux); /*Insere s na fila Q*/
  while (!eh_vazia(Q))
  {
    aux = cons_ret(Q); /*Consulta e remove o primeiro elemento da fila Q*/
    u = aux.indInf;
    v = G;
    ind = -1;
```

```
    while (v>=0)
    {
      ind++;
      if (adjacent (node, u, v))
      {
        if (cor[ind] == 'B')
        {
          DADOS aux2;
          cor[ind] = 'C';
          d[ind] = d[aux.indMemoria] + 1;
          pai[ind] = u;
          aux2.indInf = v;
          aux2.indMemoria = ind;
          ins (Q, aux2);
        }
      }
      v = node[v].next;
    }
    cor[aux.indMemoria] = 'P';
  }
}
```

# Grafos – Busca em largura

Analisaremos agora o tempo de execução do algoritmo apresentado para implementar a busca em largura sobre um grafo de entrada **G** = (**V**, **E**).

A manipulação das cores assegura que cada vértice seja colocado na fila no máximo uma vez, e portanto é retirado da fila no máximo uma vez. As operações de enfileirar e desenfileirar demoram o tempo **O**(1), e assim o tempo total dedicado a operações de filas é **O**(**V**). Pelo fato da lista de adjacências de cada vértice ser examinada somente quando o vértice é desenfileirado, a lista de adjacências de cada vértice é examinada no máximo uma vez.

# Grafos – Busca em largura

Tendo em vista que a soma dos comprimentos de todas as listas de adjacências é **O**(E), é gasto no máximo o tempo **O**(E) na varredura total das listas de adjacências. A sobrecarga correspondente à inicialização é **O**(V) e, desse modo, o tempo de execução total do algoritmo de busca em largura é **O**(V + E).

Assim, a pesquisa primeiro na extensão é executada em tempo linear no tamanho da representação de lista de adjacências de **G**.

**Observação:** Os resultados da busca em largura podem depender da ordem na qual os vizinhos de um determinado vértice são visitados; a árvore primeiro na extensão pode variar, mas as distância **d** calculadas pelo algoritmo não irão variar.

# Grafos – Busca em largura

Como determinar o caminho mínimo (menor caminho) entre um vértice A e B?

Com a busca em largura!

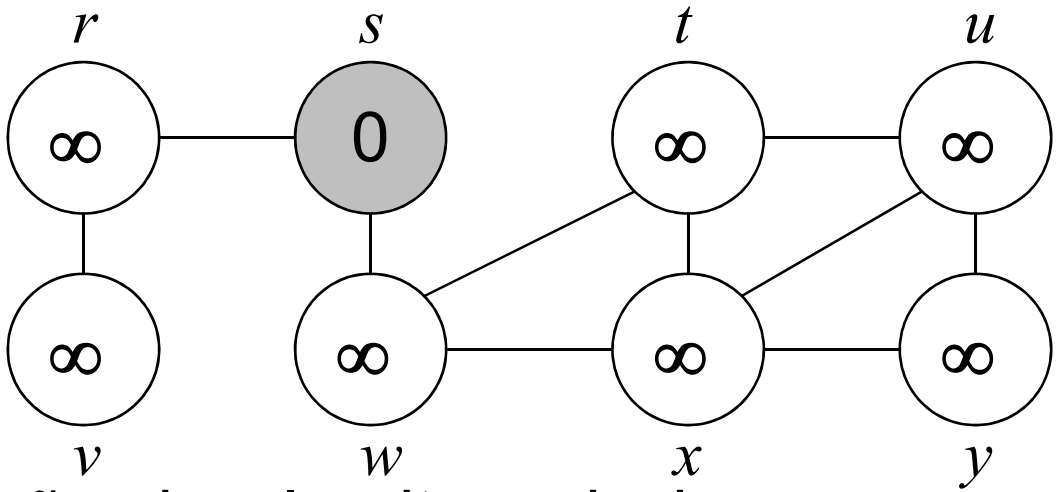
Como?

Conforme foi anteriormente mencionado, o procedimento de busca em largura constrói uma árvore primeiro na extensão à medida que pesquisa o grafo. A árvore é representada pelo campo **pai** em cada vértice. Mais formalmente, para um grafo **G** = (**V**, **E**) com origem ***s***, definimos o **subgrafo predecessor** de **G** como:

$$G_{pai} = (V_{pai}, E_{pai}), \text{ onde}$$
$$V_{pai} = \{v \in V: pai[v] \neq NULL\} \cup \{s\} .$$
e
$$E_{pai} = \{\{(pai[v], v): v \in V_{pai} - \{s\}\}.$$

# Grafos – Busca em largura

O subgrafo predecessor $\mathbf{G_{pai}}$ é uma **árvore primeiro na extensão** se $\mathbf{V_{pai}}$ consiste nos vértices acessíveis a partir de ***s*** e, para todo $\boldsymbol{v} \in \mathbf{V_{pai}}$ existe um caminho único simples desde ***s*** até ***v*** em $\mathbf{G_{pai}}$ que também é um caminho mais curto de ***s*** até ***v*** em **G**. As arestas em $\mathbf{E_{pai}}$ são chamadas arestas da árvore. Observe o grafo abaixo:




A aplicação do algoritmo da busca em profundidade gera as seguintes informações nos respectivos campos:

| | |
|---|---|
| **Vértices:** | **{ y, x, w, v, u, t, s, r }** |
| **Distâncias:** | **{ 3, 2, 1, 2, 3, 2, 0, 1 }** |
| **Pais:** | **{ x, w, s, r, t, w, , s }** |

# Grafos – Busca em largura

Com base no que foi apresentado implemente uma função, na linguagem C, que com base no resultado da busca em largura apresente o caminho mínimo de um vértice ***s*** para um vértice ***v***.

```
void imprimirCaminho(listaDeNodos node, int s, int v)
{
  if (v==s)
    printf("%c ",node[s].info);
  else
    if (node[v].pai == -1)
      printf("\nNenhum caminho de \"%c\" para \"%c\" existente.\n",node[s].info,node[v].info);
    else
    {
      imprimirCaminho(node, s, node[v].pai);
      printf("%c ",node[v].info);
    }
}
```

# Grafos – Busca em largura

Estudamos a determinação dos caminhos mais curtos para grafos não-ponderados. Porém, este processo pode ser generalizado passando a tratar situações em que cada aresta tem um valor de peso real e o peso de um caminho é a soma dos pesos de suas arestas constituintes.

Sob este prisma, os grafos considerados em nosso estudo são grafos não-ponderados ou, de modo equivalente, todas as arestas têm peso unitário.



UNIVASF

# Grafos – Busca em profundidade

Estudaremos agora a busca em profundidade. A estratégia seguida pela busca em profundidade é, como seu nome implica, procurar "mais fundo" no grafo sempre que possível.

Na busca em profundidade, as arestas são exploradas a partir do vértice ***v*** mais recentemente descoberto que ainda tem arestas inexploradas saindo dele.

Quando todas as arestas de ***v*** são exploradas, a busca "regressa" para explorar as arestas que deixam o vértice a partir do qual ***v*** foi descoberto. Esse processo continua até descobrirmos todos os vértices acessíveis a partir do vértice de origem inicial.



UNIVASF

# Grafos – Busca em profundidade

Se restarem quaisquer vértices não descobertos, então um deles será selecionado como uma nova origem, e a busca se repetirá a partir daquela origem.

Esse processo inteiro será repetido até que todos os vértices sejam descobertos.

Como ocorre no caso da busca em largura, sempre que um vértice ***v*** é descoberto durante uma varredura da lista de adjacências de um vértice já descoberto ***u***, a busca em profundidade registra esse evento definindo o campo predecessor de ***v***, o campo pai[***v***], como ***u***.



UNIVASF

# Grafos – Busca em profundidade

Diferente da busca em largura, cujo subgrafo predecessor forma uma árvore, o subgrafo predecessor produzido por uma busca em profundidade pode ser composto por várias árvores, porque a busca pode ser repetida a partir de várias origens.

O subgrafo predecessor de uma busca em profundidade é então definido de forma ligeiramente diferente daquela de uma busca em largura: fazemos $\mathbf{G_{pai}} = (\mathbf{V}, \mathbf{E_{pai}})$, onde

$$\mathbf{E_{pai}} = \{(\mathbf{pai}[v], v) : v \in \mathbf{V} \text{ e } \mathbf{pai}[v] \neq \mathbf{NULL}\}$$

# Grafos – Busca em profundidade

Pode parecer arbitrário que a busca em largura se limite apenas a uma origem, enquanto a busca em profundidade pode pesquisar a partir de várias origens. Embora em termos conceituais a busca em largura possa se dar a partir de várias origens e a busca em profundidade possa se limitar a uma origem, nossa abordagem reflete a forma como os resultados dessas buscas são normalmente usados.

Em geral a busca em largura é empregada para encontrar distâncias de caminhos mais curtos (e o subgrafo predecessor associado) a partir de uma origem dada. Com frequência, a busca em profundidade é uma sub-rotina em outro algoritmo.



UNIVASF

# Grafos – Busca em profundidade

O subgrafo predecessor de uma busca em profundidade forma uma **floresta primeiro na profundidade** composta por várias **árvores primeiro na profundidade**. As arestas em $E_{pai}$ são chamadas arestas de árvore.

Como ocorre no caso da busca em largura, os vértices são coloridos durante a busca, a fim de indicar seu estado.

Cada vértice é inicialmente branco, é acinzentado ao ser descoberto na busca, e depois é enegrecido quando sua lista de adjacências é completamente examinada.



UNIVASF

# Grafos – Busca em profundidade

Essa técnica garante que cada vértice acaba em exatamente uma árvore primeiro na profundidade, de forma que essas árvores sejam disjuntas.

Além de criar uma floresta primeiro na profundidade, a busca em profundidade também identifica cada vértice com um carimbo de tempo.

Cada vértice ***v*** tem dois carimbos de tempo: o primeiro carimbo de tempo **d**[***v***] registra quando ***v*** é descoberto pela primeira vez (e acinzentado), e o segundo carimbo de tempo **f**[***v***] registra quando a busca termina de examinar a lista de adjacências de ***v*** (e pinta ***v*** de preto).

# Grafos – Busca em profundidade

Esses carimbos de tempo são usados em muitos algoritmos de grafos e em geral são úteis no raciocínio sobre o comportamento da busca em profundidade.

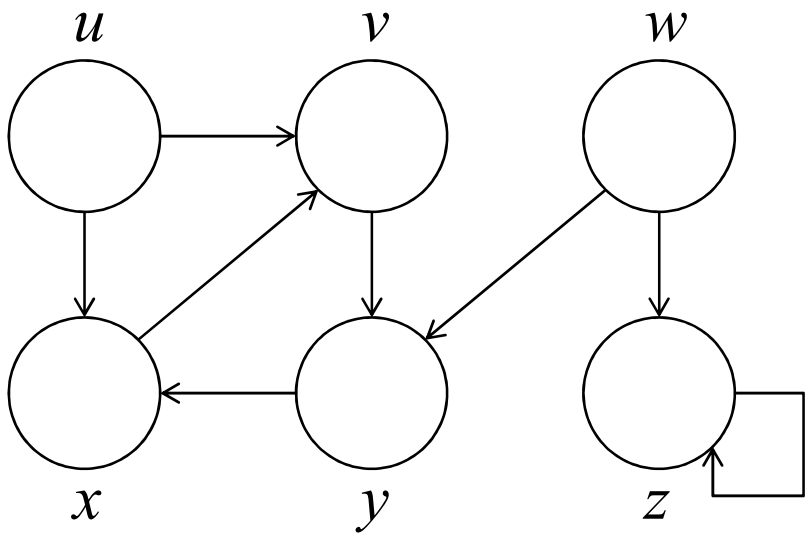
Esses carimbos de tempo são inteiros entre 1 e 2 * |V|. Pois, existe um evento de descoberta e um evento de término de examinar a lista de adjacências para cada um dos |V| vértices. Para todo vértice ***u***,

$$d[u] < f[u]$$

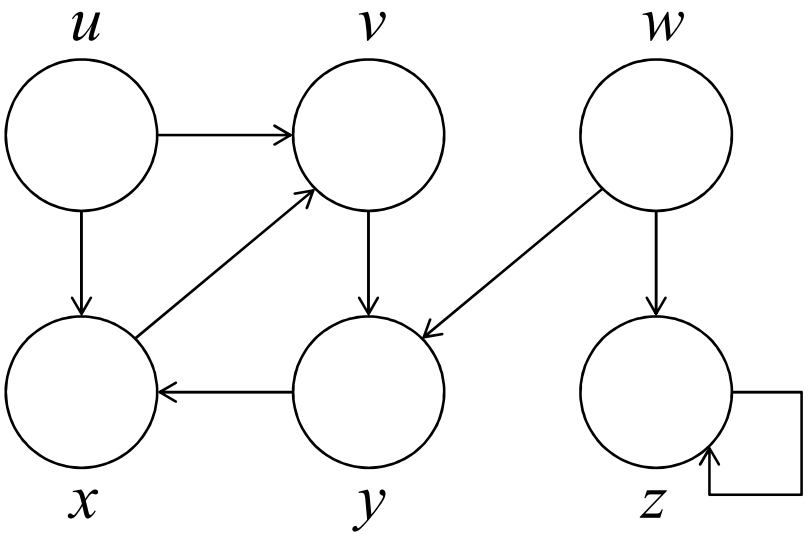
O vértice ***u*** é BRANCO antes do tempo d[***u***], CINZA entre o tempo d[***u***] e o tempo f[***u***] e PRETO depois disso.

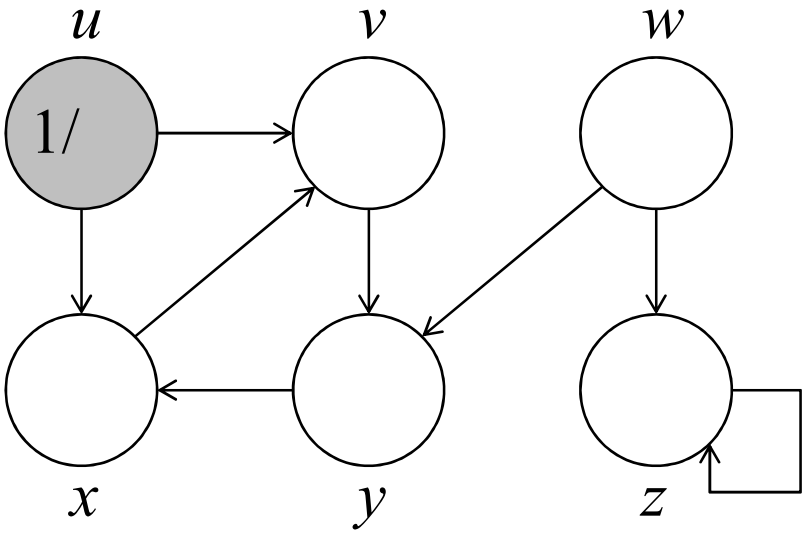
# Grafos – Busca em profundidade

Para uma melhor compreensão do processo de busca em profundidade analisaremos a aplicação do mesmo sobre o grafo a baixo.

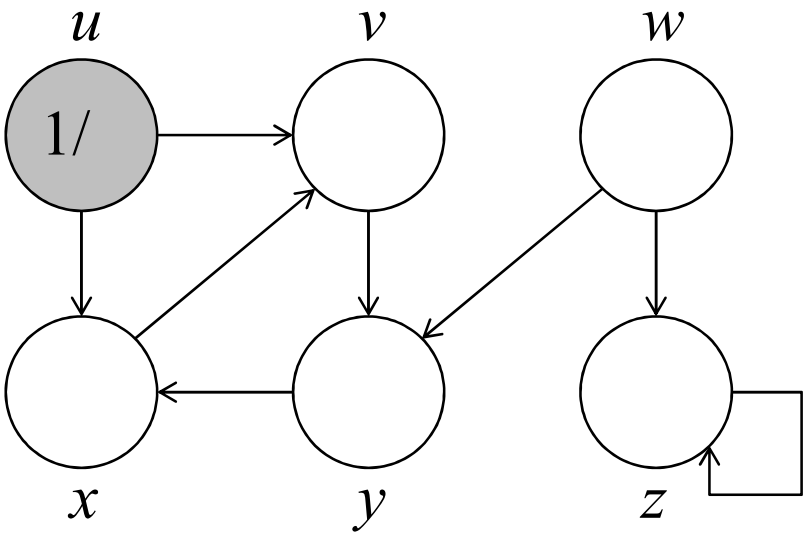


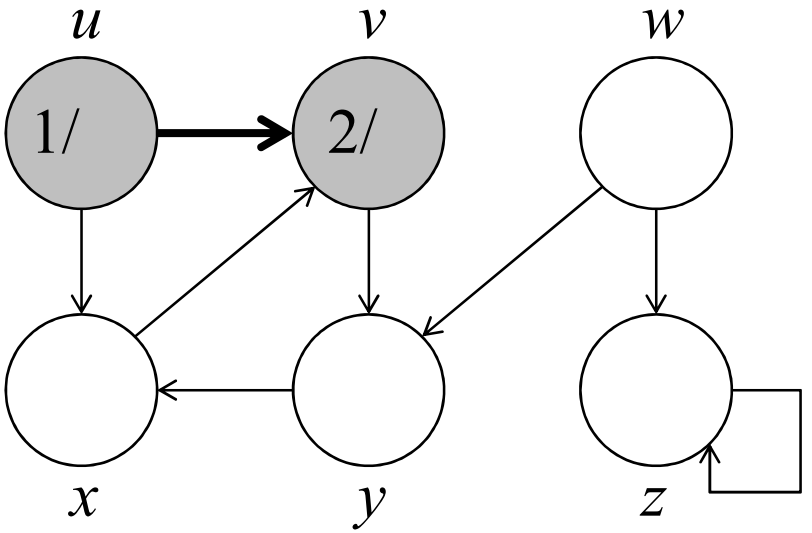
# Grafos – Busca em profundidade



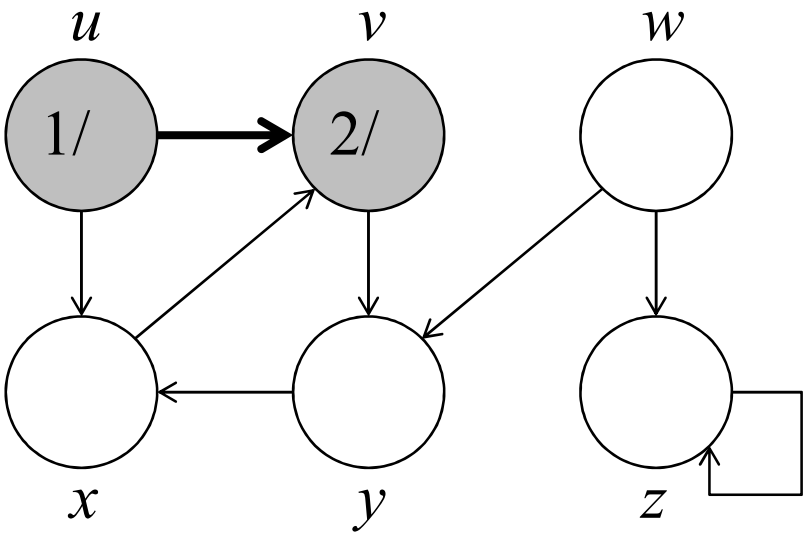



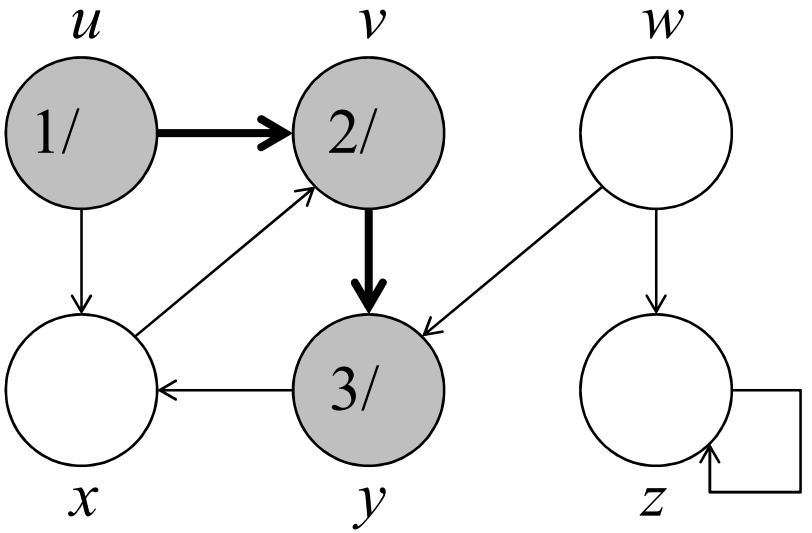
# Grafos – Busca em profundidade



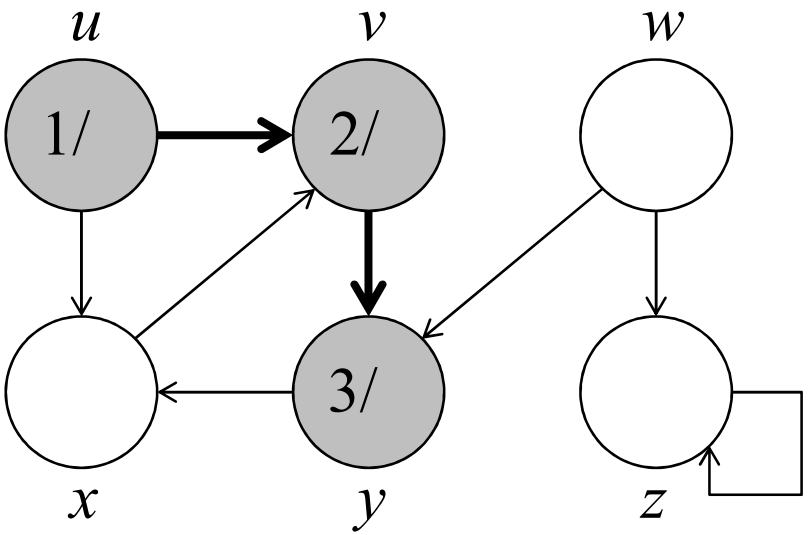



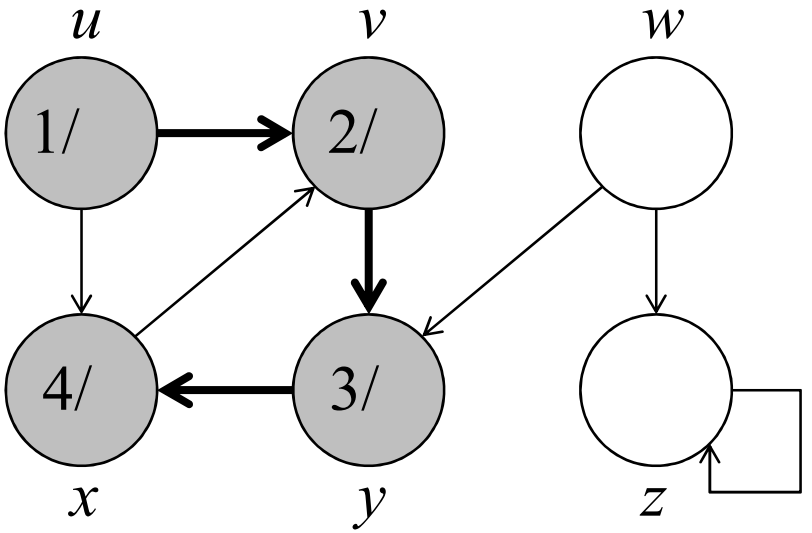
# Grafos – Busca em profundidade



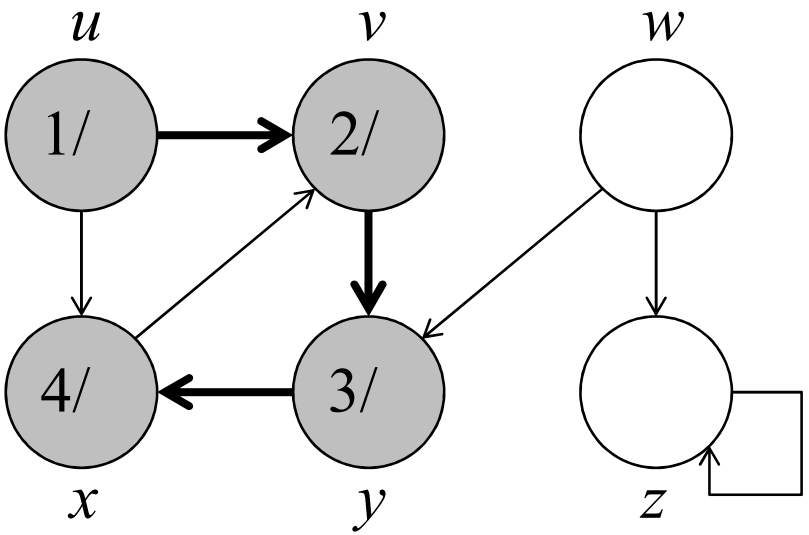



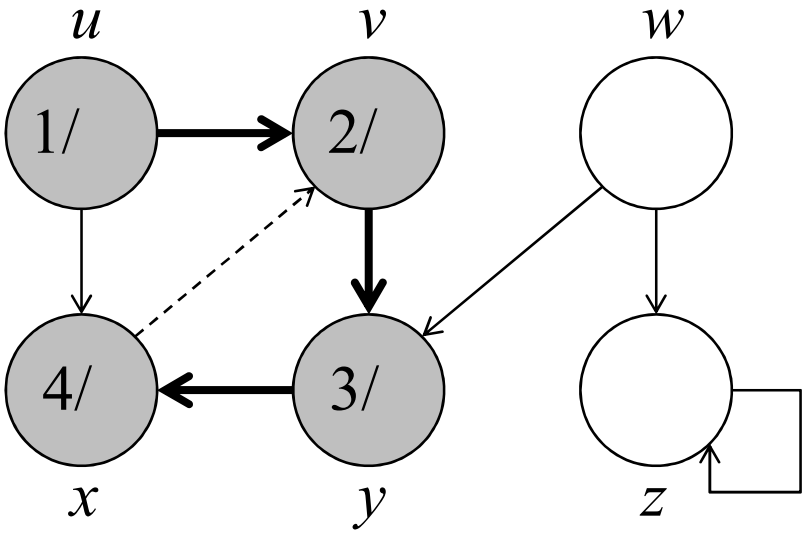
# Grafos – Busca em profundidade



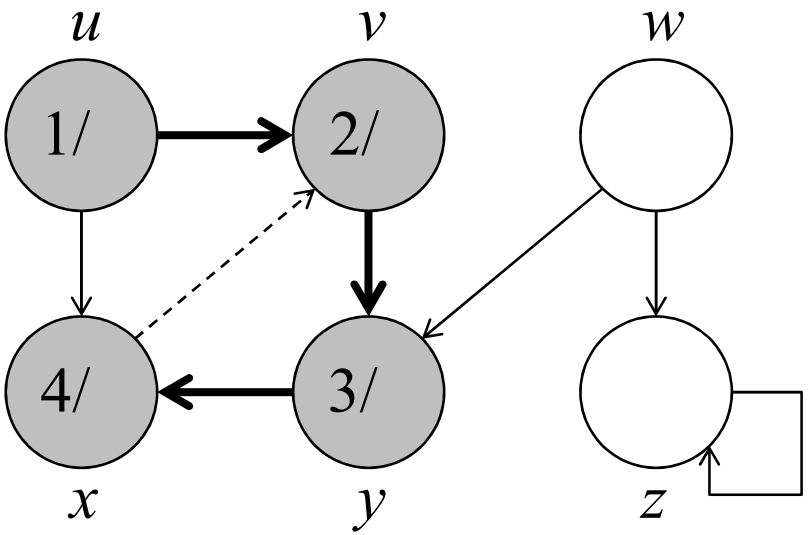



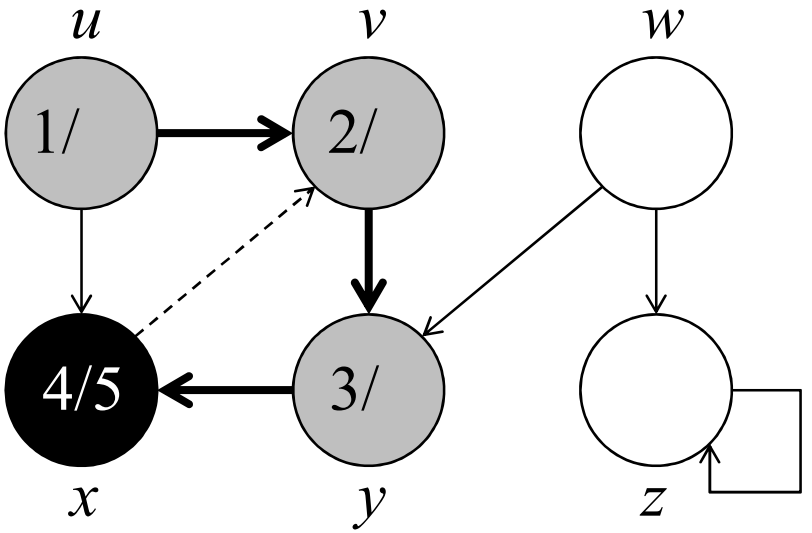
# Grafos – Busca em profundidade



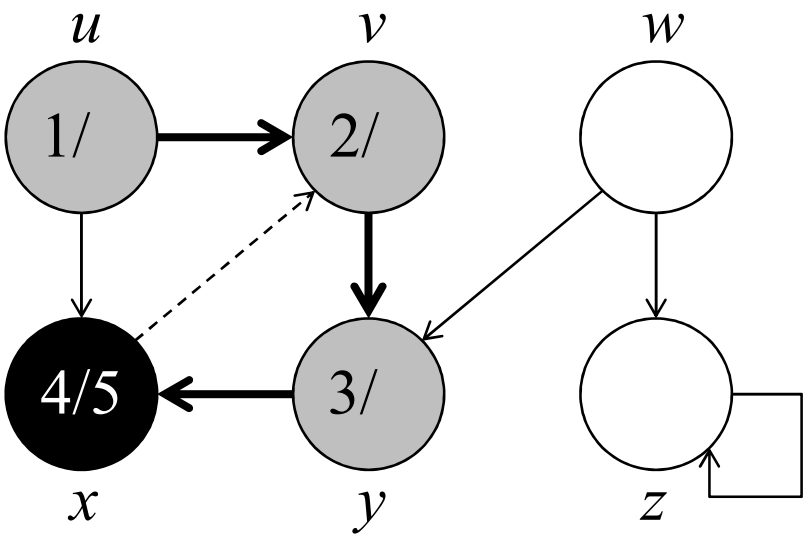



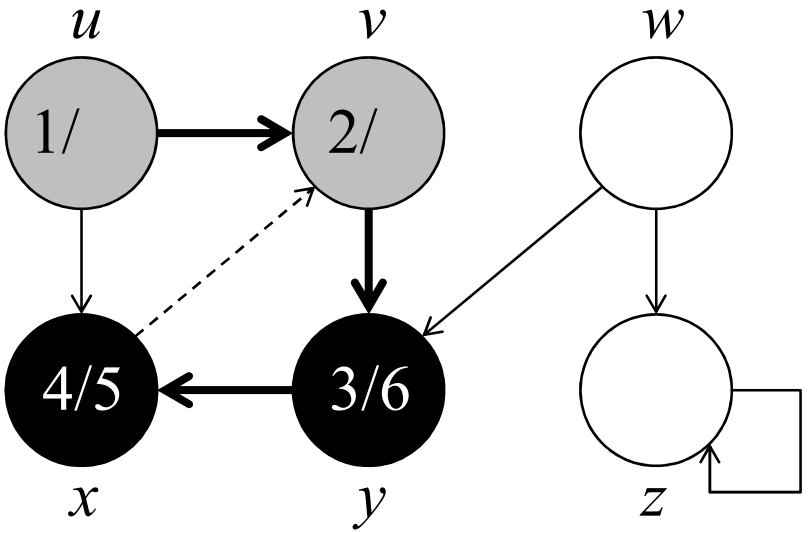
# Grafos – Busca em profundidade



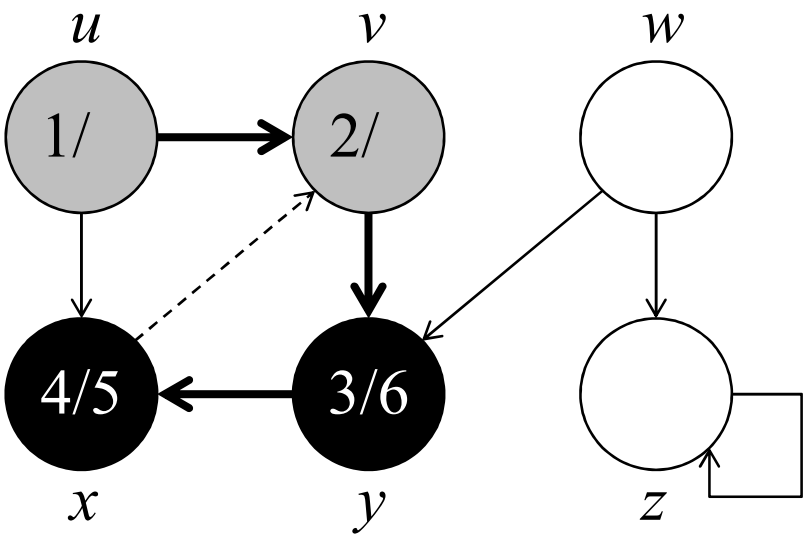



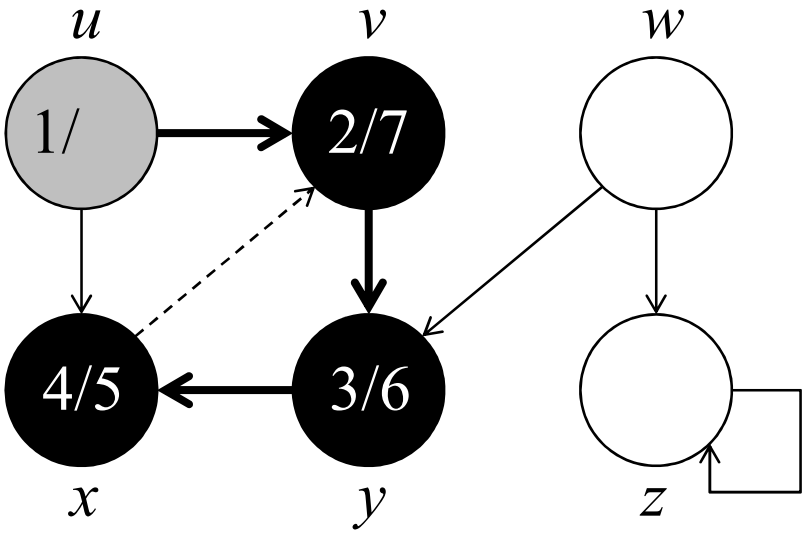
# Grafos – Busca em profundidade



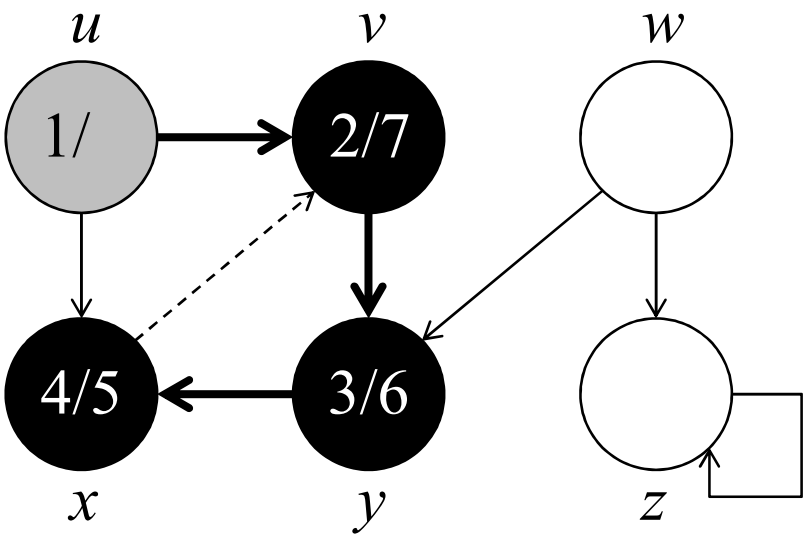



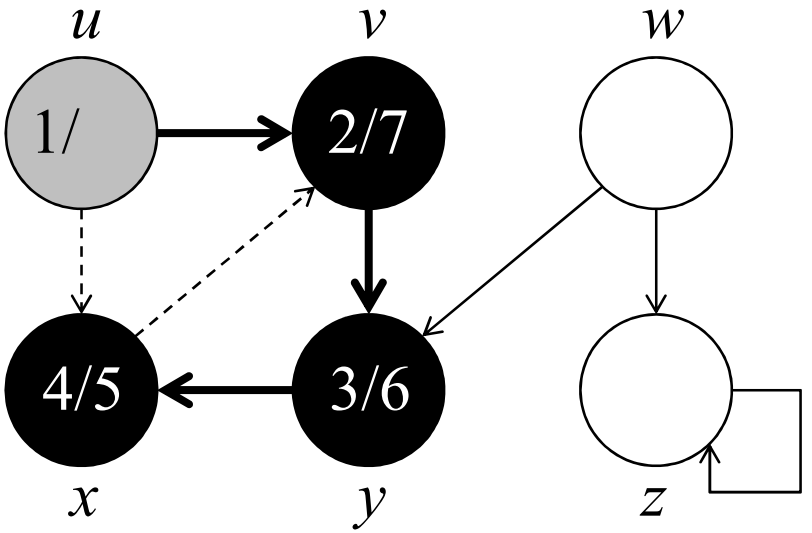
# Grafos – Busca em profundidade



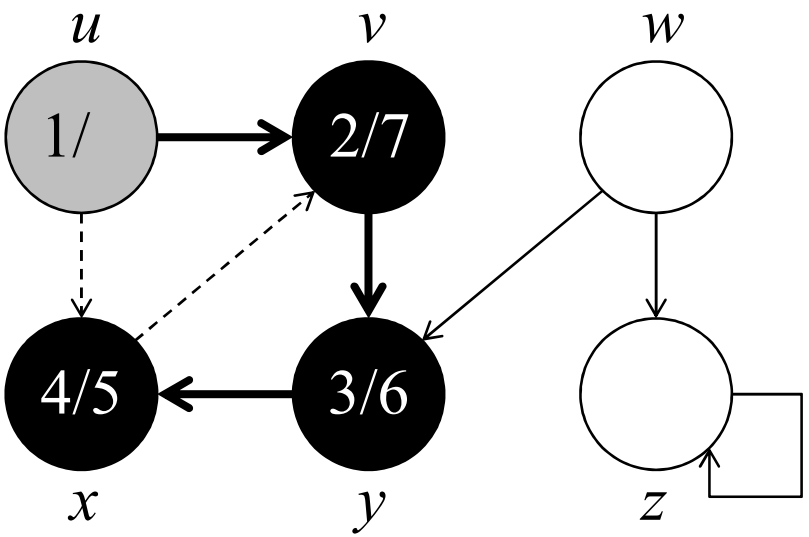



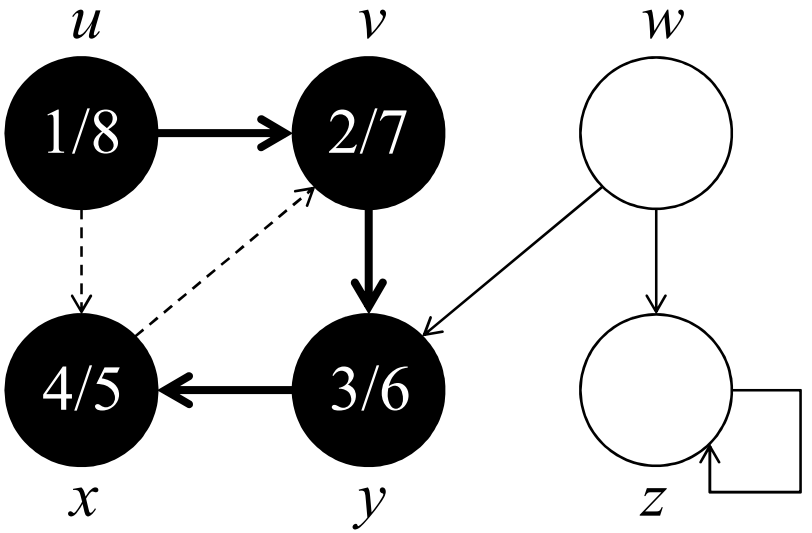
# Grafos – Busca em profundidade



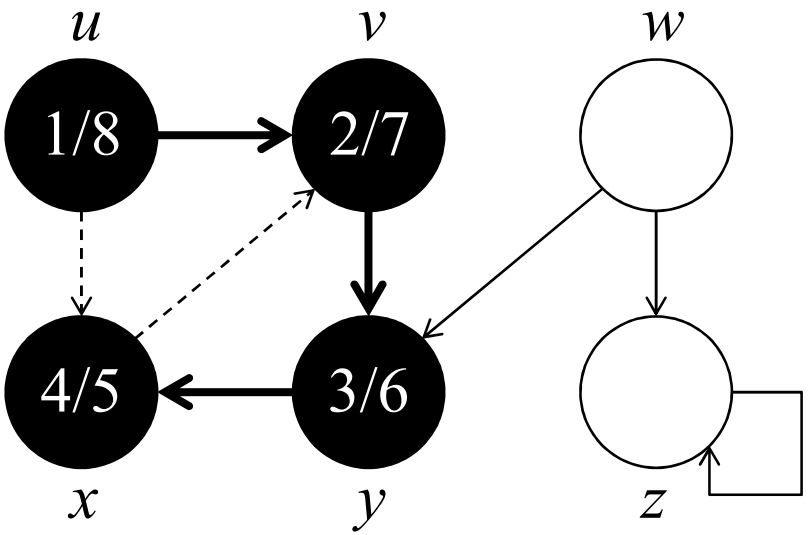



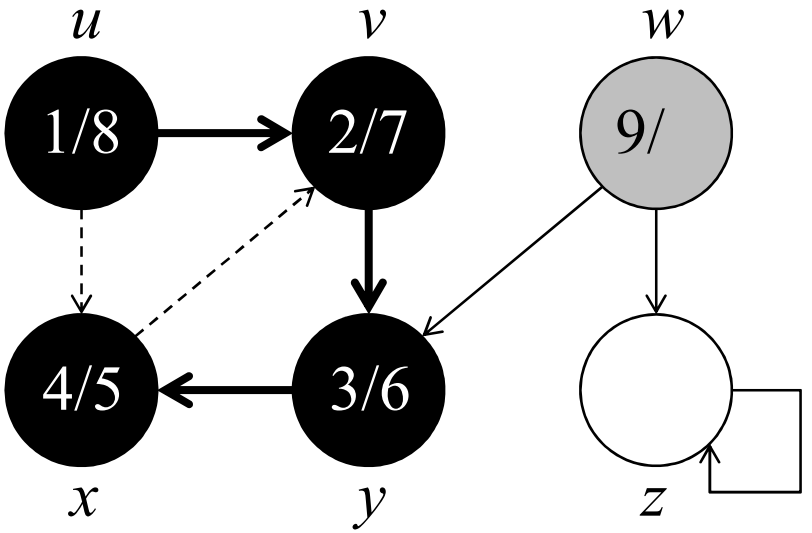
# Grafos – Busca em profundidade



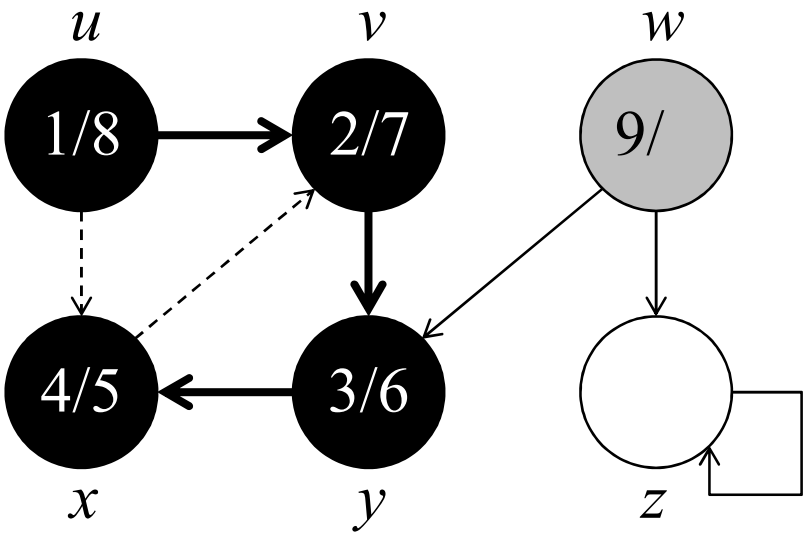



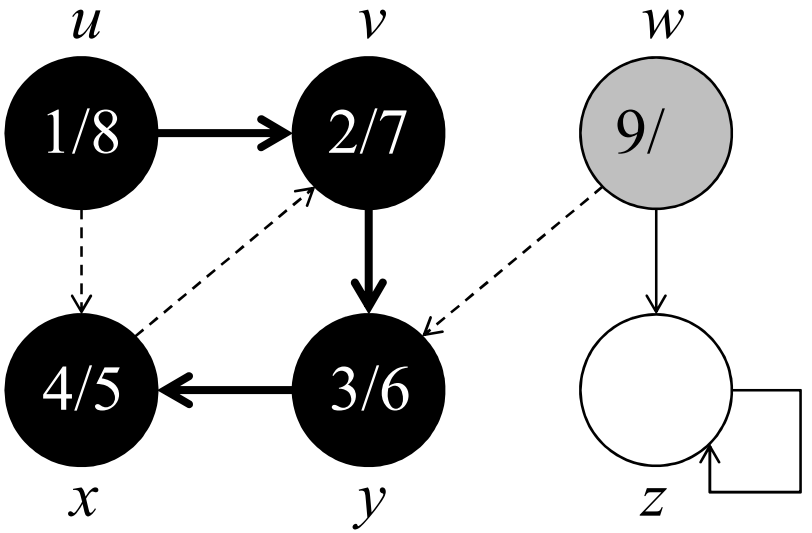
# Grafos – Busca em profundidade



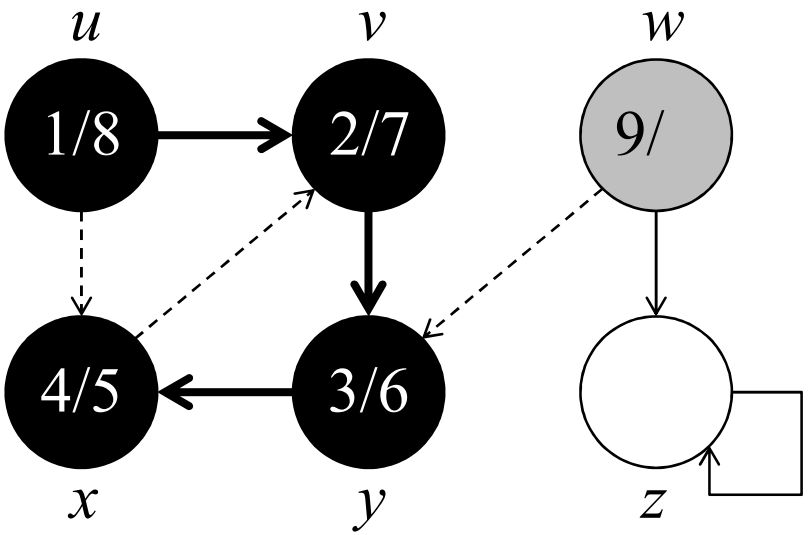



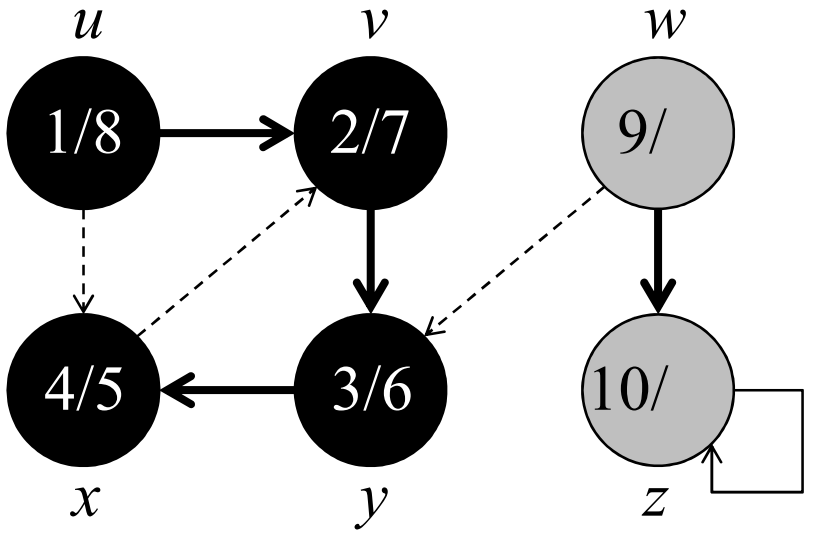
# Grafos – Busca em profundidade



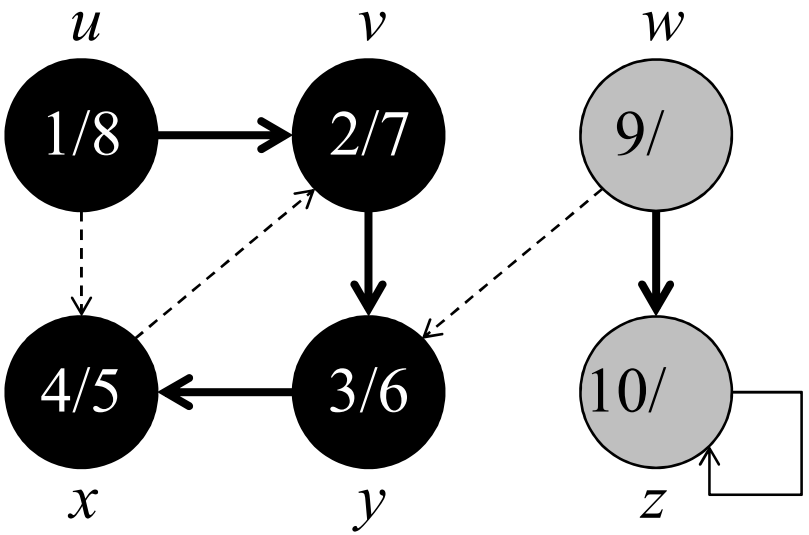



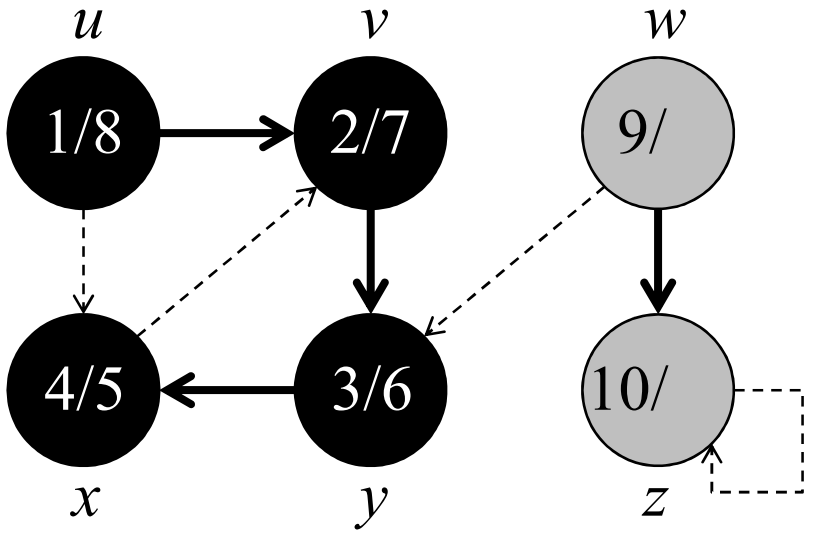
# Grafos – Busca em profundidade



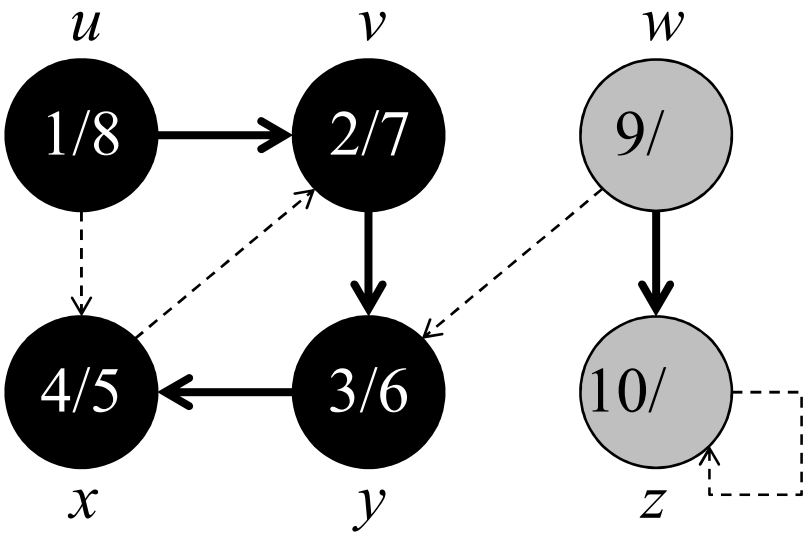



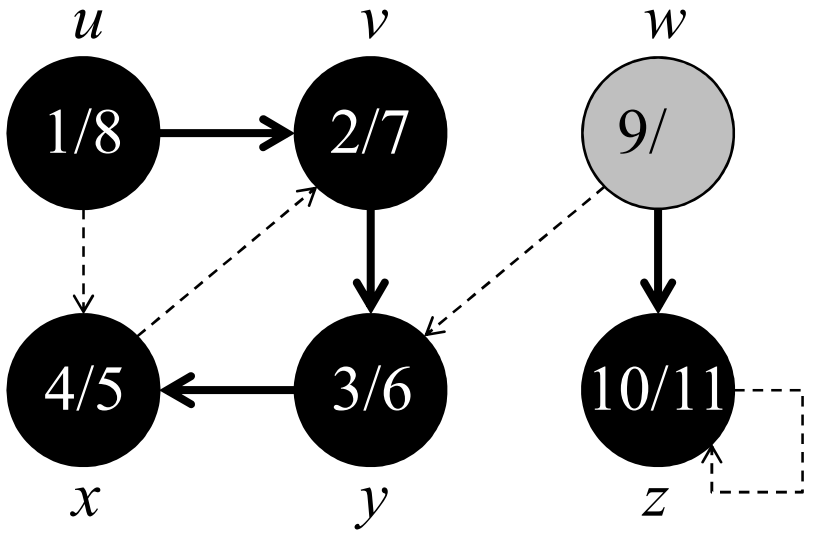
# Grafos – Busca em profundidade



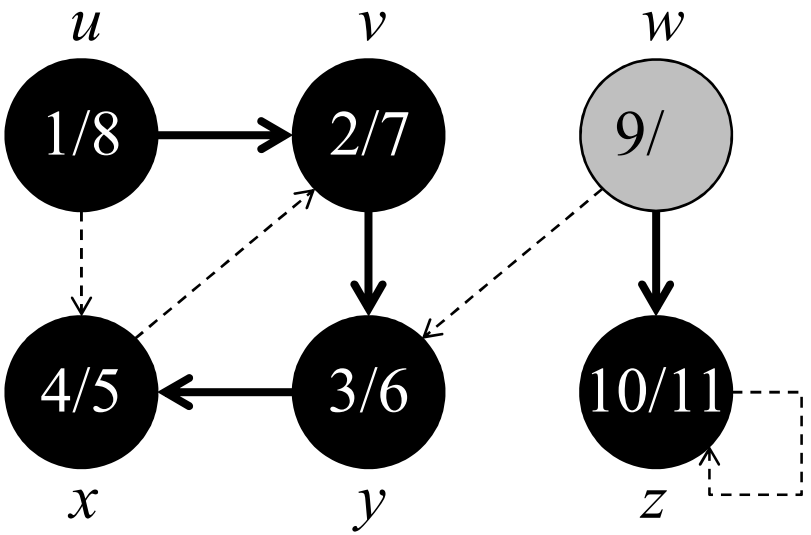



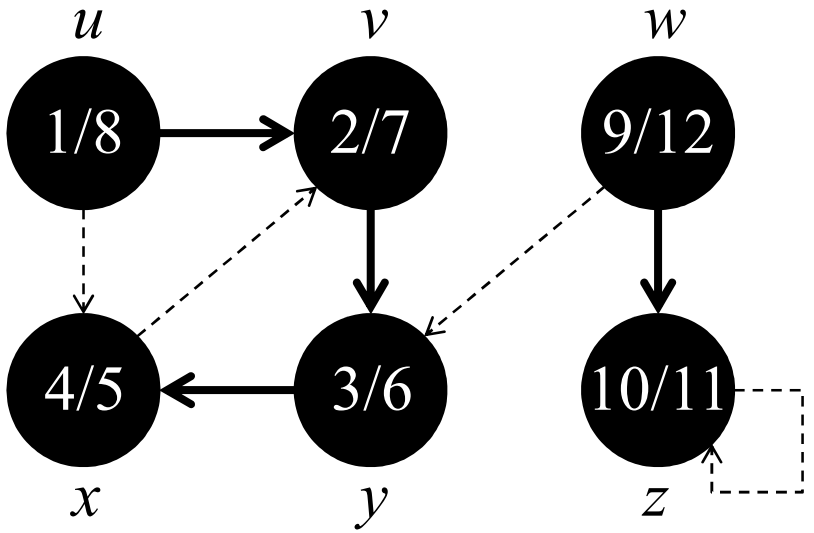
# Grafos – Busca em profundidade



# Grafos – Busca em profundidade



# Grafos – Busca em profundidade

Com base no que foi estudado, codifique uma função, na linguagem C, que implemente o procedimento de busca em profundidade discutido, pressuponha que o grafo de entrada **G** = (**V**, **E**) é representado com o uso de listas de adjacências.

**Dica**: Mantenha estruturas de dados adicionais para armazenar informações relativas a cada vértice no grafo. As informações relevantes são cor, pai, primeiro carimbo de tempo que registra quando um vértice é descoberto pela primeira vez e o segundo carimbo de tempo que registra quando a busca termina de examinar a lista de adjacências do vértice. Pressuponha a existência de uma variável global que possibilite a determinação dos carimbos de tempo.



UNIVASF

```
void buscaEmProfundidade(listaDeNodos node, int G)
{
   int u=G;
   while (u>=0)
   {
      node[u].cor = 'B';
      node[u].pai = -1;
      u = node[u].next;
   }
   tempo = 0;
   u = G;
   while (u>=0)
   {
      if (node[u].cor =='B')
         visitaBuscaEmProfundidade(node, u);
      u = node[u].next;
   }
}
```

```
void visitaBuscaEmProfundidade(listaDeNodos node, int u)
{
  int v;
  node[u].cor = 'C';
  tempo++;
  node[u].d = tempo;
  v = node[u].point;
  while (v>=0)
  {
    if (node[node[v].point].cor == 'B')
    {
      node[node[v].point].pai = u;
      visitaBuscaEmProfundidade(node, node[v].point);
    }
    v = node[v].next;
  }
  node[u].cor = 'P';
  node[u].f = ++tempo;
}
```

# Grafos – Busca em profundidade

Observe que os resultados de busca em profundidade podem depender da ordem em que os vértices são examinados e da ordem em que os vizinhos de um vértice são alcançados por visitaBuscaEmProfundidade(). Essas ordens de visitação diferentes tendem a não causar problemas na prática, pois o resultado de **qualquer** busca em profundidade pode em geral ser usado de forma eficiente, com resultados essencialmente equivalentes.

Qual é o tempo de execução do procedimento de busca em profundidade?



UNIVASF

# Grafos – Busca em profundidade

Os loops do procedimento buscaEmProfundidade() demoram o tempo **O**(V), fora o tempo para executar as chamadas a visitaBuscaEmProfundidade().

O procedimento visitaBuscaEmProfundidade() é chamado exatamente uma vez para cada vértice $v \in V$. Pois, visitaBuscaEmProfundidade() é invocado somente sobre vértices brancos e a primeira ação é pintar o vértice de cinza. Durante uma execução de visitaBuscaEmProfundidade(***v***), o loop interno é executado |Adj[v] | vezes. Tendo em vista que

$$\sum_{v \in V} |Adj[v]| = \mathbf{O}(E) ,$$

o custo total da execução de visitaBuscaEmProfundidade() é **O**(E). Portanto, o tempo de execução de buscaEmProfundidade() é **O**(V + E).

# The end!