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);
    }
}
```

UNIVASF

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

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

UNIVASF

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).

UNIVASF

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***,

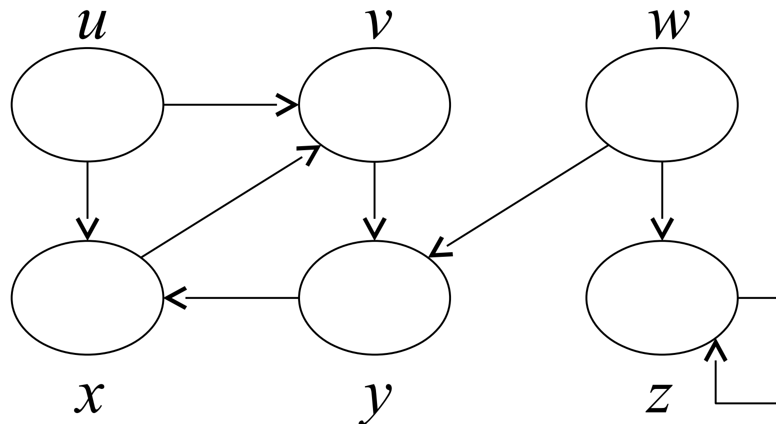
$$d[u] < f[u]$$

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.

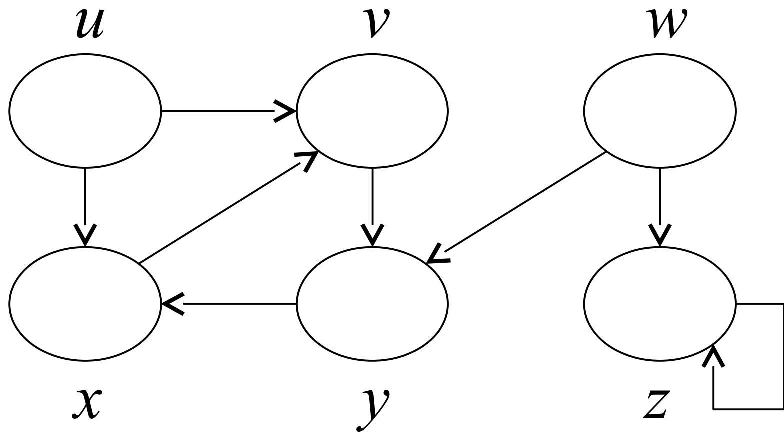
UNIVASF

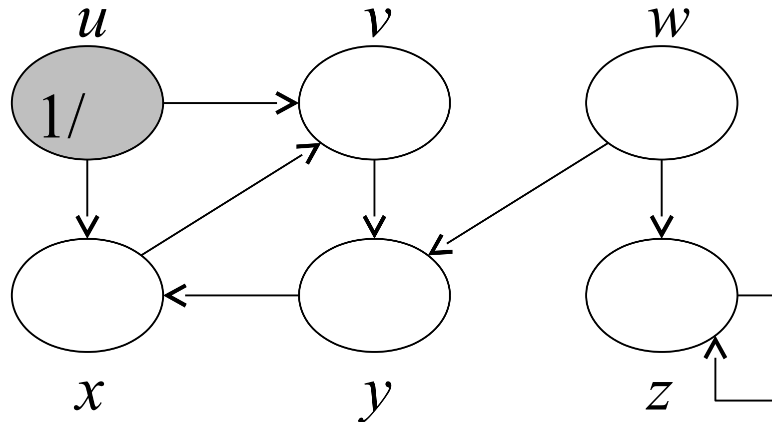
Grafos – Busca em profundidade

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

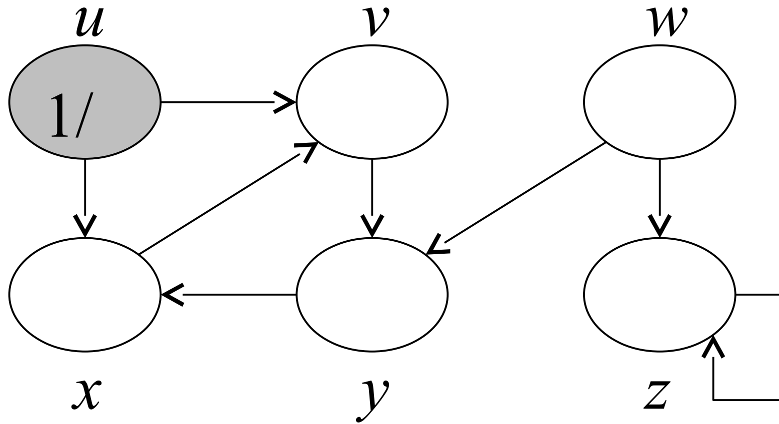


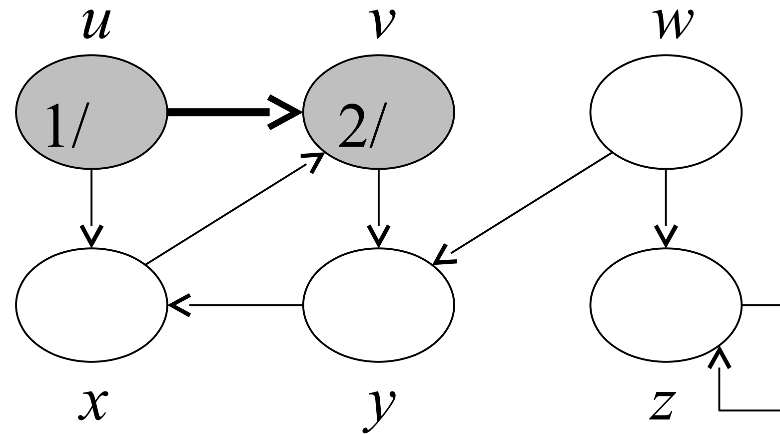
Grafos – Busca em profundidade



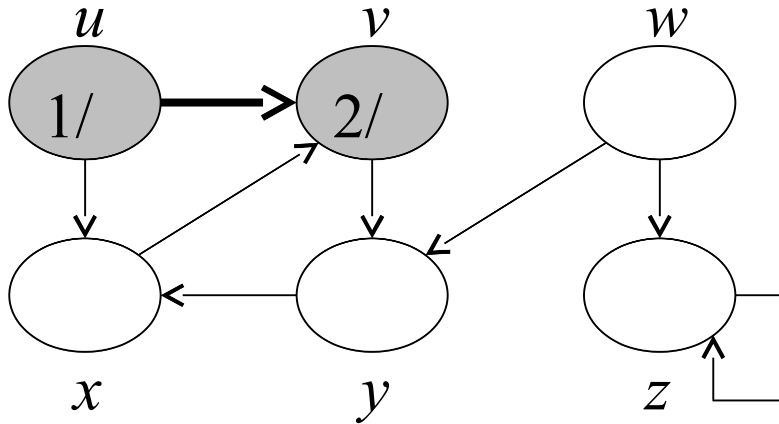


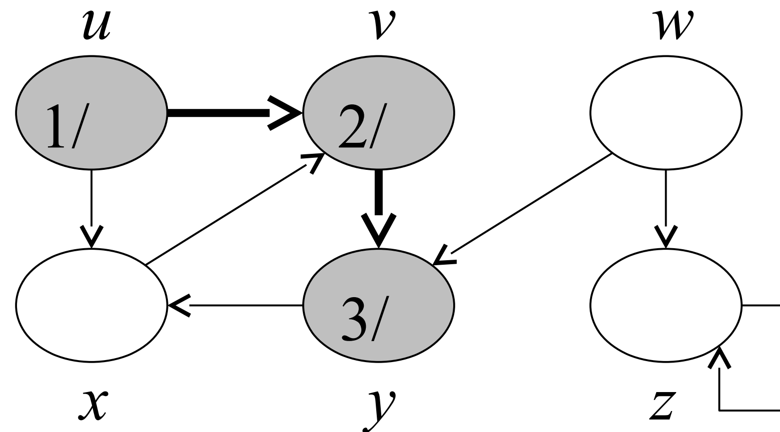
Grafos – Busca em profundidade





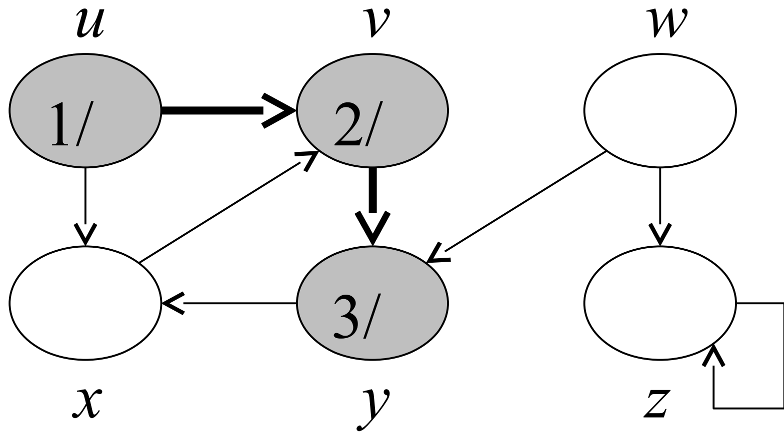
Grafos – Busca em profundidade

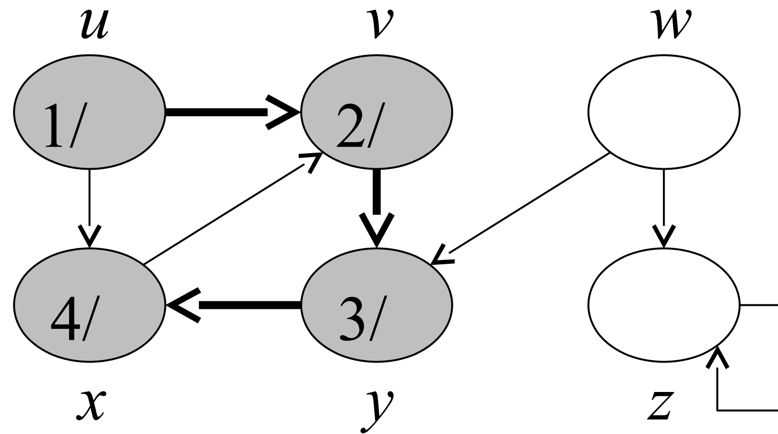




UNIVASF

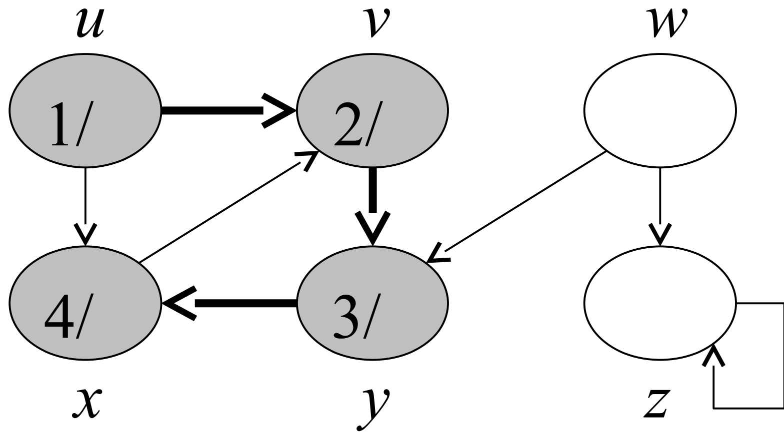
Grafos – Busca em profundidade

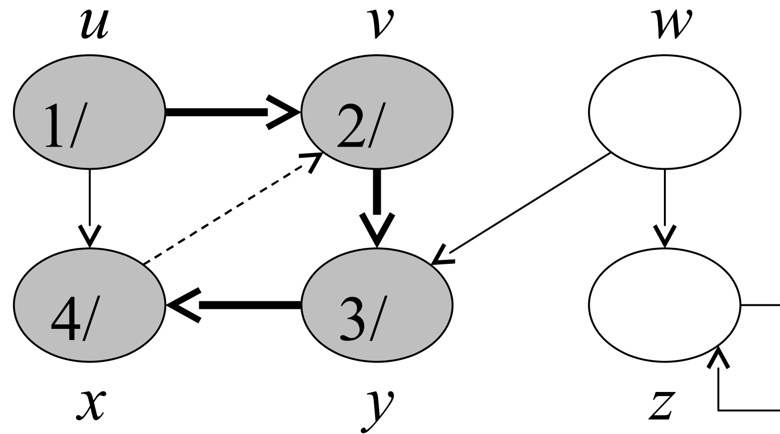




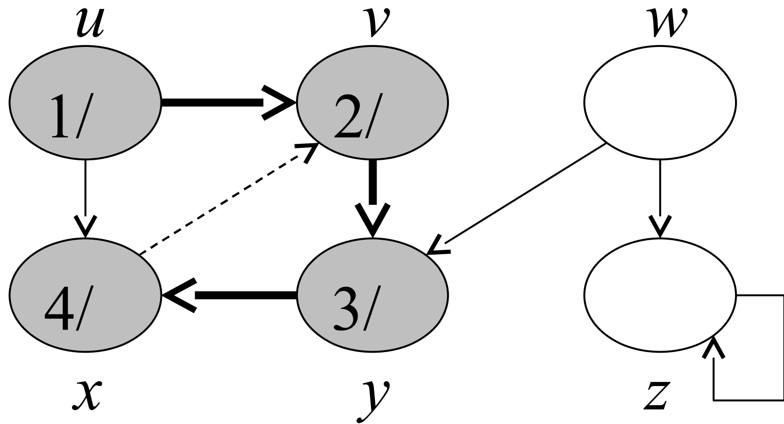
UNIVASF

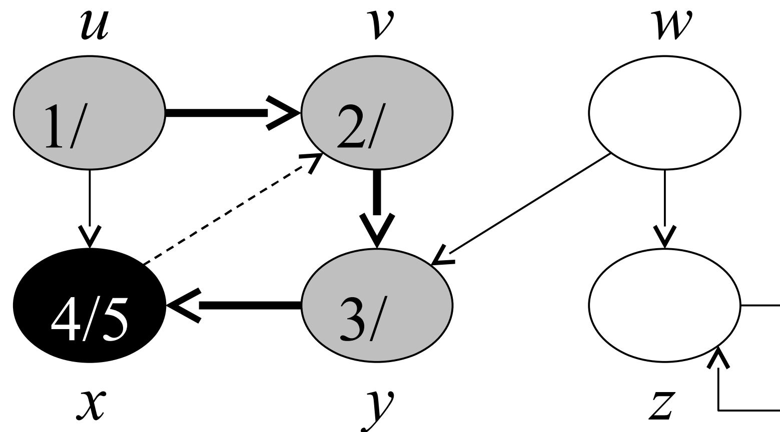
Grafos – Busca em profundidade



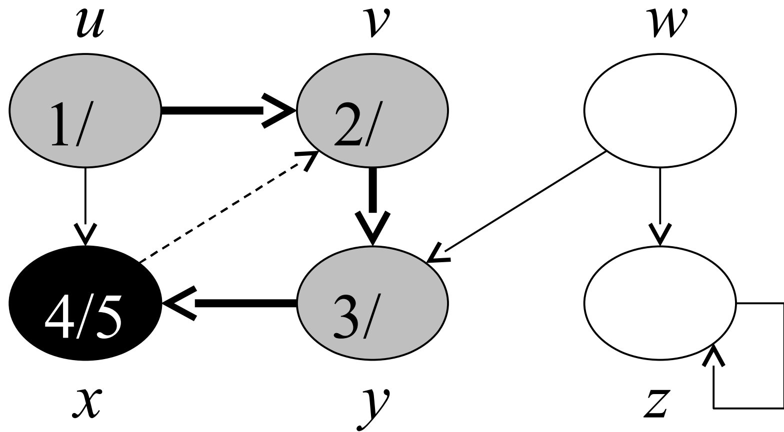


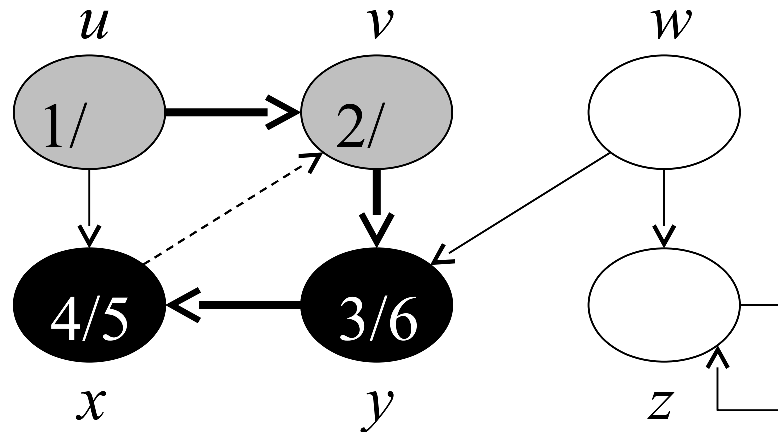
Grafos – Busca em profundidade



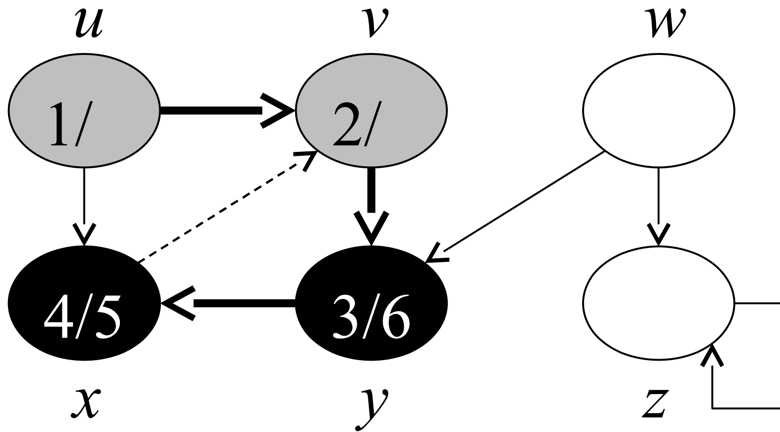


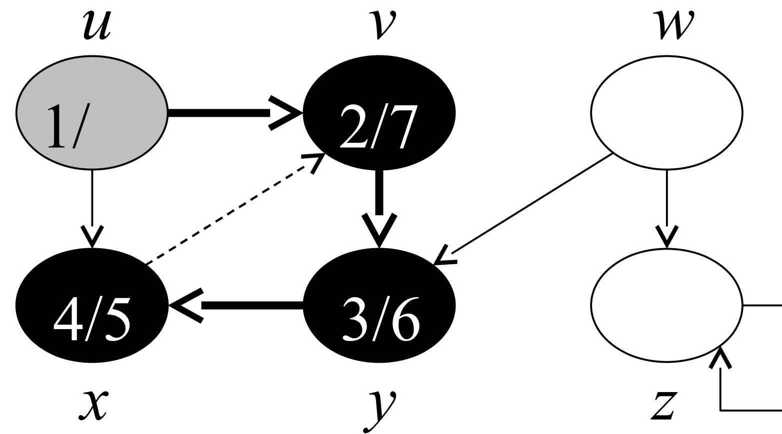
Grafos – Busca em profundidade





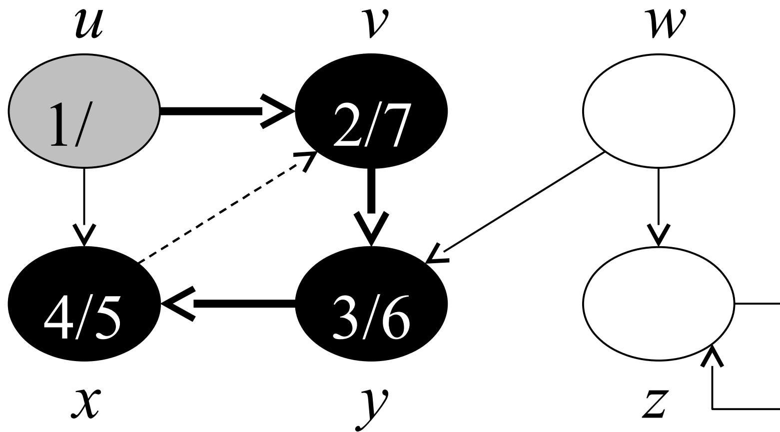
Grafos – Busca em profundidade

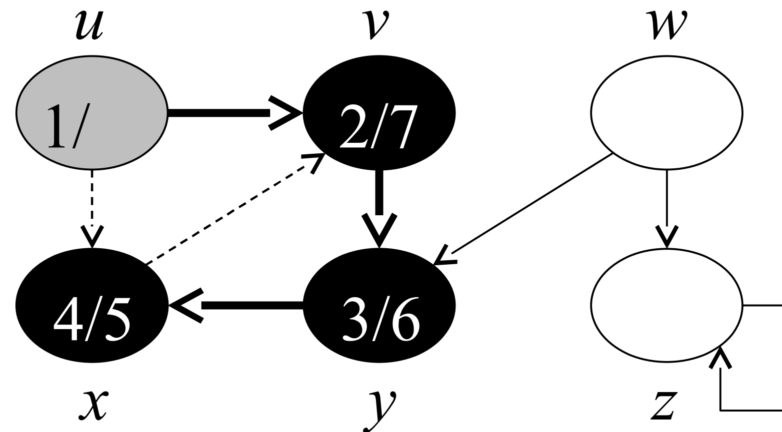




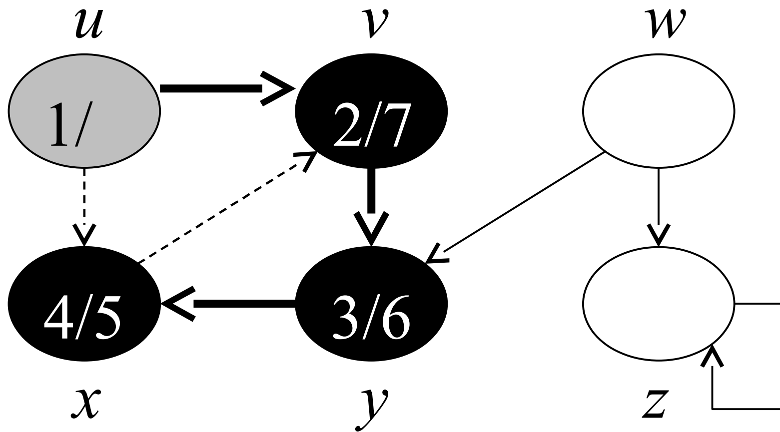
UNIVASF

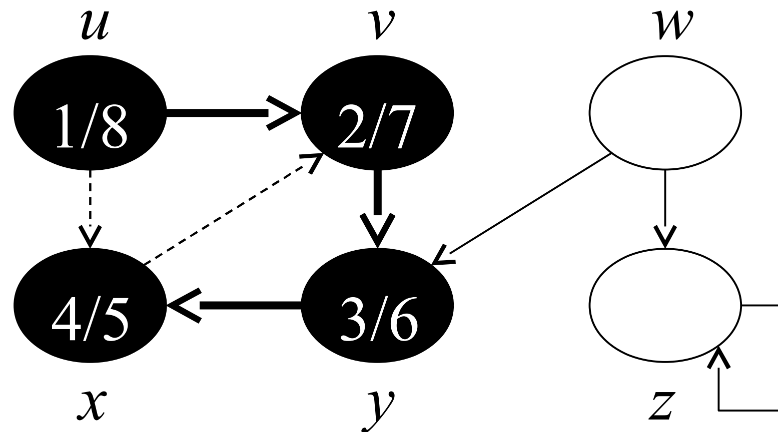
Grafos – Busca em profundidade



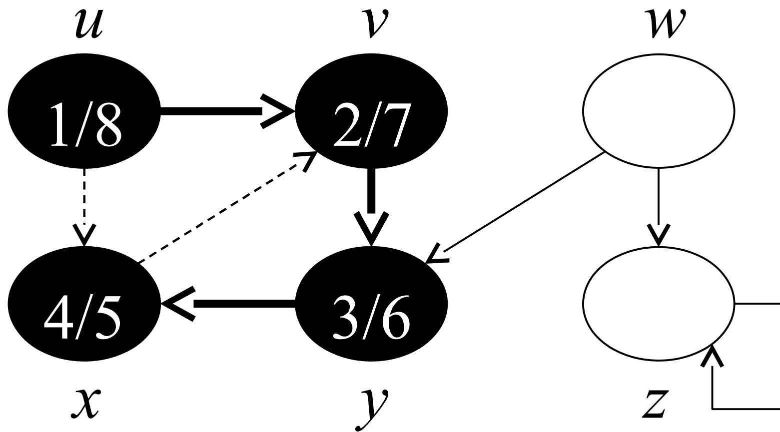


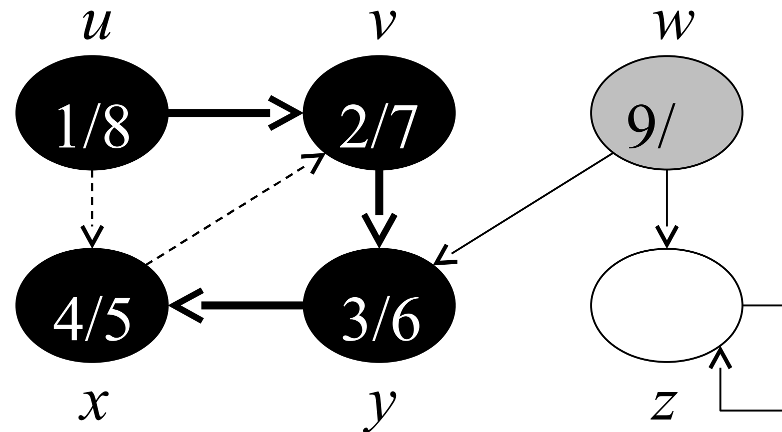
Grafos – Busca em profundidade





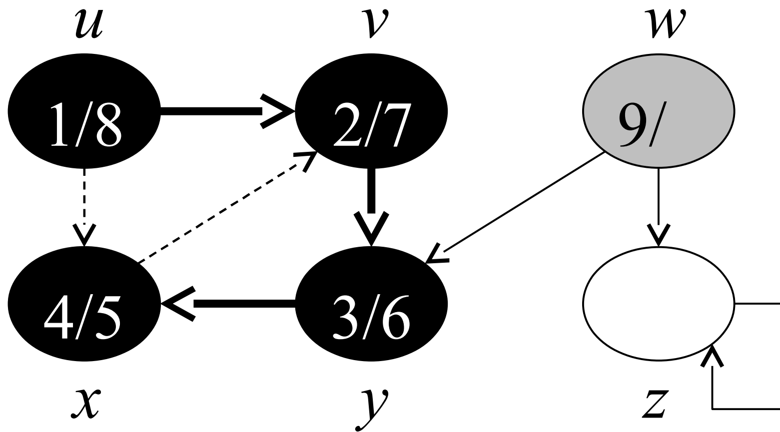
Grafos – Busca em profundidade

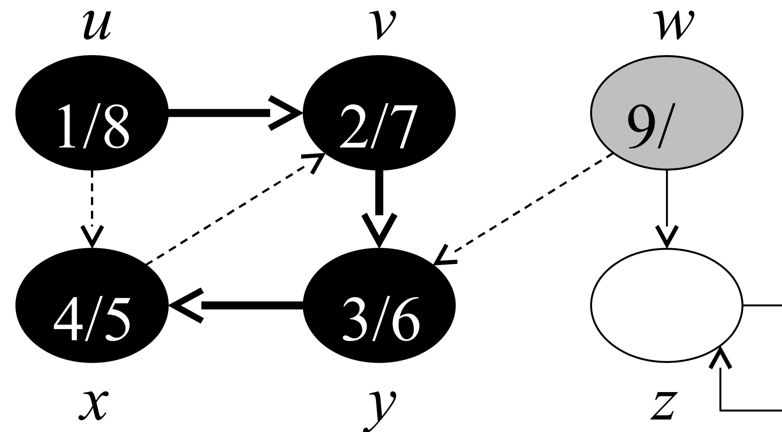




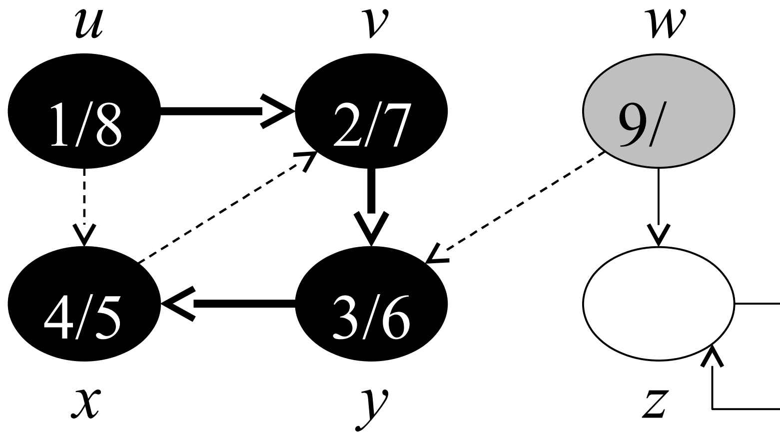
UNIVASF

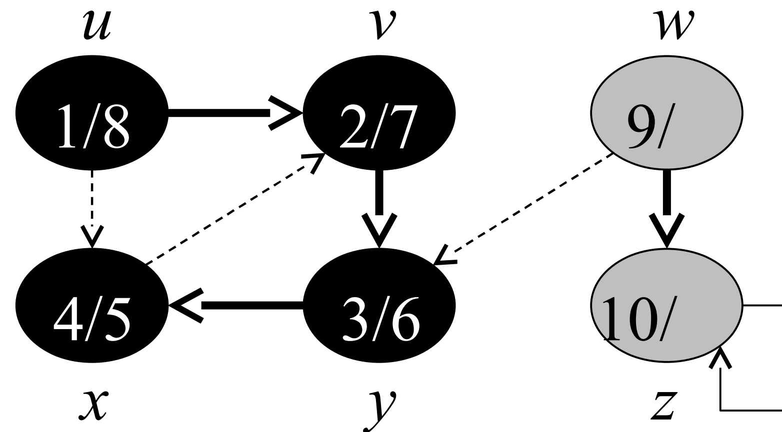
Grafos – Busca em profundidade



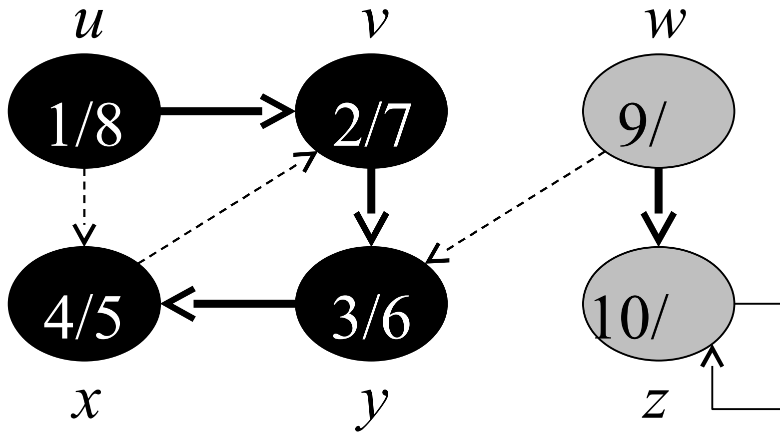


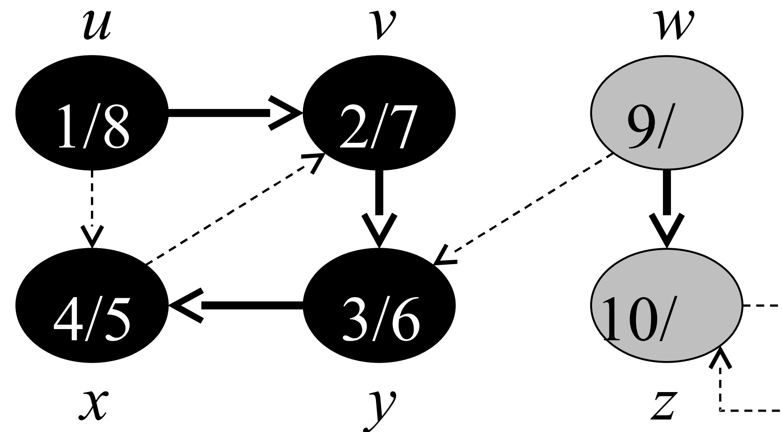
Grafos – Busca em profundidade



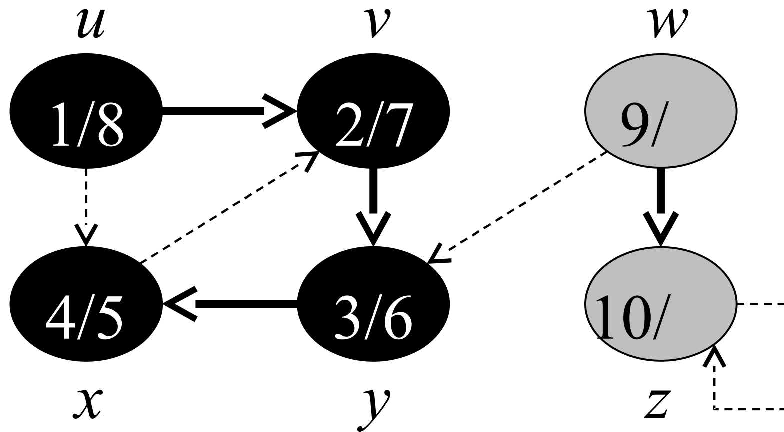


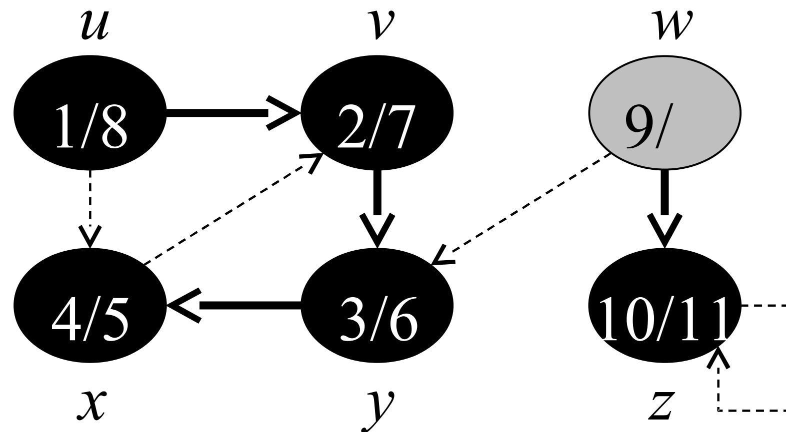
Grafos – Busca em profundidade





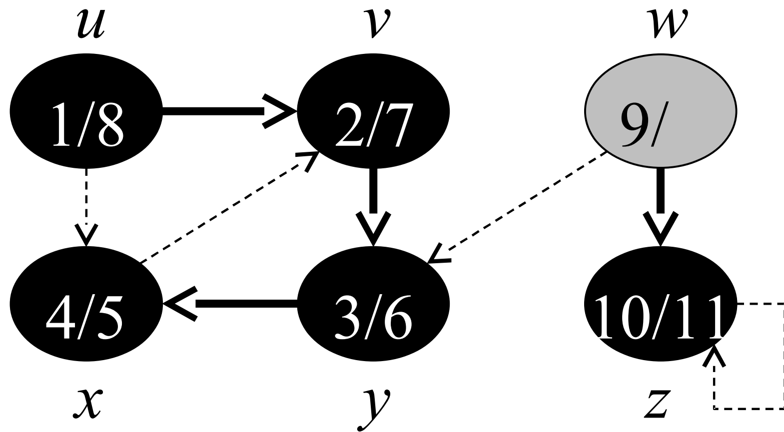
Grafos – Busca em profundidade

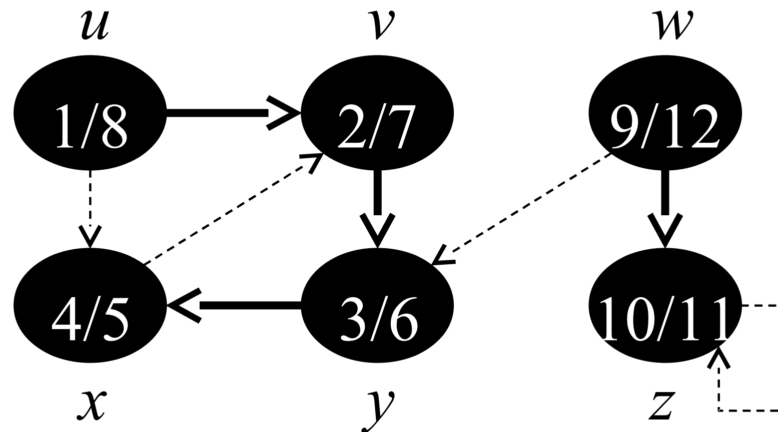




UNIVASF

Grafos – Busca em profundidade





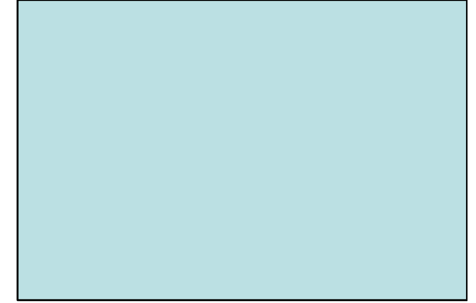
Grafos – Busca em profundidade

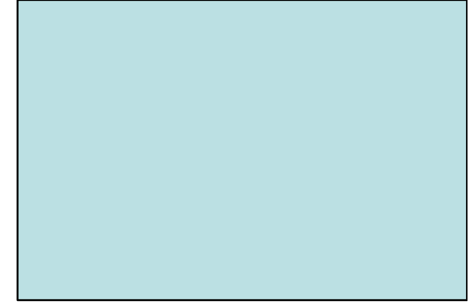
Com base no que foi estudado, codifique uma função, na linguagem C, que implemente o procedimento de busca em profundidade discutido, pressuponha que o grafo de entrada **G** = (**V**, **E**) é representado com o uso de listas de adjacências.

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; /* variável global */
  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?

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 lAdj[v] l 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).

UNIVASF

Introdução à complexidade
de algoritmos

Tempo de Execução

A avaliação de desempenho de um algoritmo quanto executado por um computador pode ser feita a posteriori ou a priori.

Uma avaliação a posteriori envolve a execução propriamente dita do algoritmo, medindo-se o tempo de execução. Só podendo ser exata se forem conhecidos detalhes da arquitetura da máquina, da linguagem de programação usada, do código gerado pelo compilador, etc. De fato, o tempo deve ser medido fisicamente para um certo algoritmo, compilador e computador. Obtêm-se assim medidas até certo pondo empíricas, ainda que guiadas por conjuntos de testes preparados para tal, chamados *benchmarks*.

UNIVASF

Tempo de Execução

Já uma avaliação a priori de um algoritmo, feita sem sua execução, de forma analítica, é possível se considerarmos dois itens:

- a entrada (os dados fornecidos) e
- o número de instruções executadas pelo algoritmo.

Em geral, o aspecto importante da entrada é seu "tamanho", que pode ser dado como número de valores num vetor, o número de registros num arquivo, enfim, um certo número de elementos que constituem a entrada de dados para o algoritmo. De modo que o tempo de execução de um algoritmo pode ser dado como uma função **T(n)** do tamanho **n** da sua entrada.

UNIVASF

Tempo de Execução

Por exemplo, um programa pode ter tempo de execução **$T(n) = n^2 + n + 1$**. A unidade de **T(n)** é em princípio instrução executada. Uma instrução neste contexto é uma sequência de operações cujo tempo de execução pode ser considerado constante (de certa forma, cada "passo" do algoritmo).

Por exemplo, o algoritmo a abaixo:

```
void somaVetor (int v[n], int *k) {
    int i;
    *k=0;
    for (i=0; i<n; i++)
        *k = (*k) + v[i];
}
```

Tempo de Execução

Sendo **v** a entrada, de tamanho **n**, pode-se ver facilmente que a soma **k + v[i]** será efetuada **n** vezes. Consequentemente, **T(n) = n+1**, incluindo o passo de inicialização. O tempo de execução (em instruções) variará conforme variar **n**, numa proporção linear.

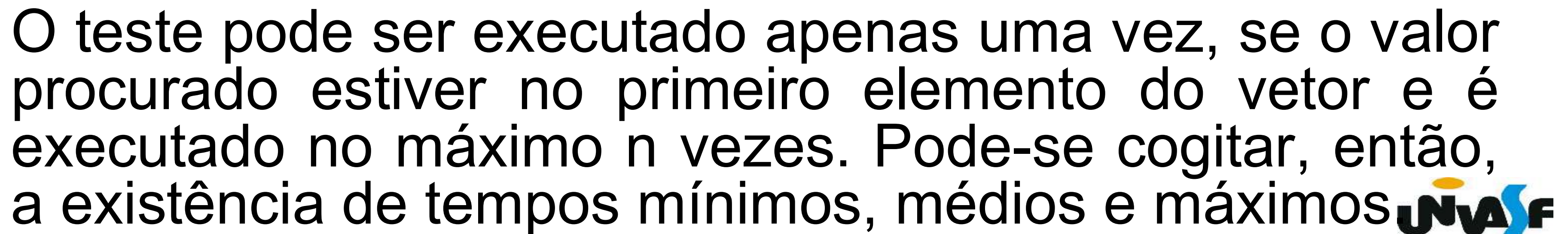
No entanto, como já se frisou, o tempo de execução vai depender de outros fatores, ligados à máquina, linguagens usadas, etc., e mesmo, por vezes, é função de aspectos adicionais de uma particular entrada e não apenas do seu tamanho, conforme veremos no exemplo do slide a seguir.

Conforme Ziviani (2004), é importante enfatizar que **T(n)** não representa diretamente o tempo de execução, mas o número de vezes que certa operação relevante é executada.

UNIVASF

Tempo de Execução

```
int localiza (int v[n], int x)
 {
    int i;
    for (i=0; i<n; i++)
       if (x==v[i])
          return i;
    return -1;
 }
```



O teste pode ser executado apenas uma vez, se o valor procurado estiver no primeiro elemento do vetor e é executado no máximo n vezes. Pode-se cogitar, então, a existência de tempos mínimos, médios e máximos.

Tempo de Execução

Acontece que, em geral, tempos mínimos são de pouca utilidade e tempos médios são difíceis de calcular (dependem de se conhecer a probabilidade de ocorrência das diferentes entradas para o algoritmo).

Consideraremos **T(n)** como uma medida assintótica máxima, ou seja, uma medida do ***pior caso*** de desempenho, que ocorre com a entrada mais desfavorável possível. No caso anterior, **T(n) = n + 1**, incluindo o passo de retorno, que sempre acontece uma vez.

UNIVASF

Tempo de Execução

Porém, a título ilustrativo, vamos analisar o problema de se efetuar uma busca sequencial em um vetor (função localiza). O melhor caso da busca sequencial em um vetor ocorre quando o valor a ser procurado é o primeiro a ser consultado. Já o pior caso ocorre quando o valor procurado é o último a ser consultado ou quando o mesmo não encontra-se no vetor.

Sendo assim temos:

Melhor caso: $T(n) = 1$

Pior caso: $T(n) = n$

UNIVASF

Tempo de Execução

Para determinar o tempo do caso médio, consideraremos $\mathbf{p_i}$ a probabilidade de localizar o *i*-ésimo valor. Como para encontrar o *i*-ésimo valor são necessárias ***i*** comparações, temos que:

$$\mathbf{T(n)} = 1 \cdot \mathbf{p_1} + 2 \cdot \mathbf{p_2} + 3 \cdot \mathbf{p_3} + \ldots + n \cdot \mathbf{p_n}$$

Considerando que a probabilidade de cada um dos valores ser encontrado é 1/n, temos que:

$$\mathbf{T(n)} = 1 \cdot 1/n + 2 \cdot 1/n + 3 \cdot 1/n + \ldots + n \cdot 1/n$$
$$\mathbf{T(n)} = 1/n \cdot (1 + 2 + 3 + \ldots + n) \qquad \text{(PA)}$$
$$\mathbf{T(n)} = 1/n \cdot (n \cdot (n + 1))/2$$
$$\mathbf{T(n)} = (n + 1)/2$$

UNIVASF

Complexidade

A avaliação analítica de um algoritmo pode ser feita com vistas a se obter uma estimativa do esforço de computação, não em termos de unidade de tempo propriamente, mais em termos de uma *taxa de crescimento* do tempo de execução em função do "tamanho do problema", i.e., do tamanho da entrada.

Um exemplo típico desse relacionamento entre tamanho da entrada e tempo de processamento é o caso dos algoritmos de ordenação: nestes, os dados são vistos sempre como sequências de valores com um certo comprimento, e é natural concluir que quanto maior esse sequência, mais tempo consumirá a ordenação.

UNIVASF

Complexidade

Mas, *quanto mais* tempo?

Vamos considerar o comportamento de dois algoritmos, **A1** e **A2**, que realizam a mesma tarefa em tempos $\mathbf{T_{A1}}$ e $\mathbf{T_{A2}}$, para uma entrada de tamanho **n**. Teremos então os seguintes tempos de execução para diferentes tamanhos da entrada:

TAMANHO DA ENTRADA	TEMPO ALG. A1	TEMPO ALG. A2
n	T_{A1}	T_{A2}
2n	$2T_{A1}$	$4T_{A2}$
3n	$3T_{A1}$	$9T_{A2}$
4n	$4T_{A1}$	$16T_{A2}$

Complexidade

Conclusão: ao se multiplicar o tamanho da entrada por **k**, o tempo de **A1** cresceu segundo **k** e o de **A2** segundo **k²**. Ou seja, a taxa de crescimento do tempo de execução de **A1** é proporcional a **n**, e a de **A2**, proporcional a **n²**.

Essa taxa de crescimento proporcional é chamada ***complexidade*** do algoritmo. Ela permite uma classificação dos algoritmos segundo sua categoria de complexidade e permite também comparar qualitativamente algoritmos diferentes que realizam a mesma tarefa. Podendo ser considerada em termos de tempo de execução (**complexidade de tempo**) ou termos de espaço de memória utilizado (**complexidade de espaço**).

Complexidade

Como já discutimos para o tempo de execução, pode-se ter complexidade de melhor, médio e pior caso. Em nosso estudo, a necessidade de espaço em memória será considerada constante e o termo *complexidade* designará *complexidade de tempo de pior caso.*

A expressão da complexidade de um algoritmo busca refletir suas condições intrínsecas, abstraindo aspectos ligados aos ambientes específicos de execução. Assim, não são consideradas constantes de soma ou multiplicação. Uma expressão como $\mathbf{k \cdot n + c}$, onde **k** e **c** são constantes, deve ser simplificada para **n**.

UNIVASF

Complexidade

Outra condição que se assume é de que estamos considerando o comportamento assintótico do algoritmo, ou seja: a complexidade expressa uma *tendência a um limite* à medida que cresce o tamanho do problema. Supõe-se que a quantidade de dados a ser processada é suficientemente grande para essa tendência se evidenciar.

Isto leva a outra simplificação: ao se ter uma expressão polinomial **P(n)**, como os termos de menor grau podem ser desprezados (quando **n** é grande) diante do termo de maior grau, este é que será adotado como aproximação. Por exemplo: $an^3 + bn^2 - cn + d$, onde **a**, **b**, **c** e **d** são constantes, será reduzido a n^3.

UNIVASF