PCS 2215

Fundamentos de Engenharia de Computação II

Aulas 6-11

Árvores

Edson Satoshi Gomi

Professor Responsável

versão: 1.2 (ago 2002)

Conteúdo

1. Terminologia e caracterização das árvores.
2. "Spanning trees".
3. Árvores binárias.
4. Métodos de percurso de árvores.
5. Árvores de decisão.

Introdução

- Árvores formam uma das subclasses de grafos mais utilizadas.
- Árvores são utilizadas, por exemplo, na organização e relacionamento de dados em Banco de Dados.

Definição : árvore

Definição : uma <u>árvore (livre)</u> T é um grafo simples que satisfaz a seguinte condição:

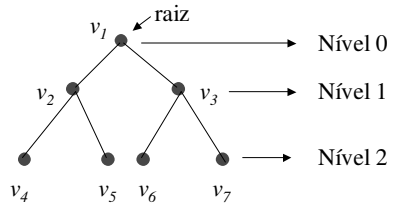
"se v e w são vértices em T, então existe um único caminho simples de v a w."

Uma <u>árvore com raiz</u> é uma árvore na qual um vértice particular foi designado como raiz.

Definições : nível de um vértice e altura de uma árvore

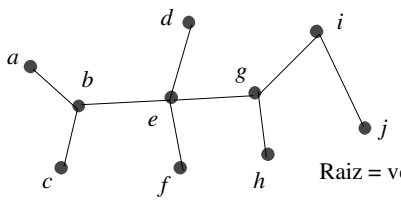
- O nível de um vértice v é o comprimento do caminho simples desde a raiz até o vértice v.
- O nível da raiz é igual a 0 (zero).
- A altura de uma árvore é igual ao número correspondente ao maior nível que ela possui.

Exemplo de árvore



Exercício

Determinar os níveis dos vértices e a altura da árvore a seguir:



Raiz = vértice e

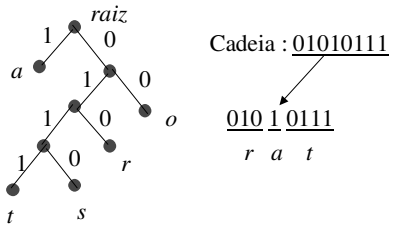
Exemplo de utilização de árvores : Código de Huffman

- Geralmente caracteres são representados por códigos formados por um número fixo de bits (por exemplo, código ASCII).
- Códigos de Huffman permitem representar caracteres através de um número variável de bits.

Código de Huffman - motivação

- A idéia é utilizar menos bits para os caracteres mais utilizados e mais bits para os caracteres menos utilizados.
- Assim, em geral é possível representar cadeias de caracteres (strings), tais como textos e programas, utilizando-se menos bits que um código de comprimento fixo.

Código de Huffman - exemplo



Código de Huffman - algoritmo

- Huffman desenvolveu um algoritmo para construir um código de Huffman a partir de uma tabela contendo a frequência de ocorrências dos caracteres a serem representados, tal que o código construído permite representar cadeias de caracteres em espaço mínimo, desde que as frequências dos caracteres de tais cadeias sejam idênticas às da tabela utilizada na construção do código. A demonstração de que o código construído é otimizado pode ser encontrada em [2].

Algoritmo para construção de um código de Huffman otimizado

Input: a sequence of n frequencies ($N \geq 2$).
Output: a rooted tree that defines an optimal Huffman code.
procedure *huffman(f,n)*
  if $n = 2$ **then**
    begin
      let f_1 and f_2 denote the frequencies;
      let T be as in figure 7.1.9;
    end
  else
    let f_i and f_j denote the smallest frequencies;
    replace f_i and f_j in the list by $f_i + f_j$;
    T' := *huffman(f, n-1);*
    replace vertex in T' labeled $f_i + f_j$ by the tree
      shown in figure 7.1.10 to obtain tree T;
  end;
  return T;
end *huffman;*

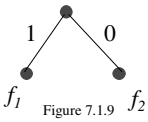


Figure 7.1.9

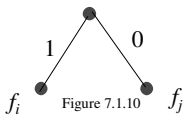
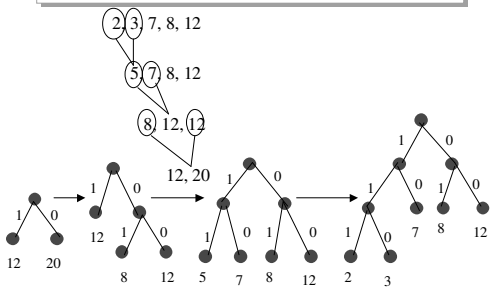


Figure 7.1.10

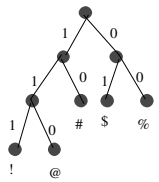
Exemplos de caracteres e respectivas frequências

Caracter	Frequência
!	2
@	3
#	7
$	8
%	12

Exemplo de execução do algoritmo



Árvore resultante



Terminologia de árvores (1)

Considere uma árvore T com raiz v_0 . Suponha que x, y e z sejam vértices em T e que $(v_0 , v_1 , ... , v_n)$ seja um caminho simples em T. Então:

- v_{n-1} é o pai de v_n .
- $v_0 , v_1 , ... , v_{n-1}$ são os ancestrais de v_n .
- v_n é um filho de v_{n-1} .
- Se x é um ancestral de y, então y é descendente de x.

Terminologia de árvores (2)

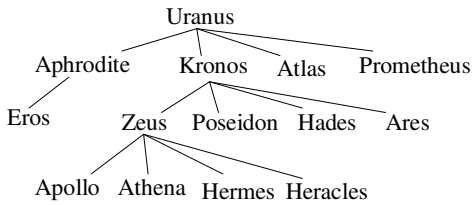
- Se x e y são filhos de z, então x e y são irmãos.
- Se x não tem filho, então x é um vértice terminal (ou uma folha).
- Se x não é um vértice terminal, então x é vértice interno.

Terminologia de árvores (3)

- A sub-árvore de T com raiz em x é o grafo com conjunto de vértices V e conjunto de arestas E, onde V é composto por x e seus descentes e

$E = \{ e \mid e$ é uma aresta pertencente a um caminho simples de x a um vértice em $V \}$.

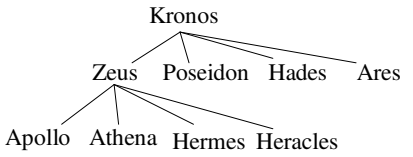
Exemplo - Família dos Deuses Gregos



Exercício : aplicando a terminologia ao exemplo

- O pai de Zeus é Kronos.
- Os ancestrais de Hermes são Zeus, Kronos e Uranus.
- Os filhos de Zeus são Apollo, Athena, Hermes e Heracles.
- Os descendentes de Kronos são Zeus, Poseidon, Hades, Ares, Apollo, Athena, Hermes e Heracles.
- Aphrodite e Prometheus são irmãos.
- Os vértices terminais são Eros, Apollo, Athena, Hermes, Heracles, Poseidon, Hades, Ares, Atlas e Prometheus.
- Os vértices internos são Uranus, Aphrodite, Kronos e Zeus.

Sub-árvore do exemplo anterior, com raiz em Kronos



Teorema

Seja *T* um grafo com *n* vértices.

As afirmações seguintes são equivalentes:

- *T* é uma árvore.
- *T* é conectado e acíclico.
- *T* é conectado e tem *n - 1* arestas.
- *T* é acíclico e tem *n - 1* arestas.

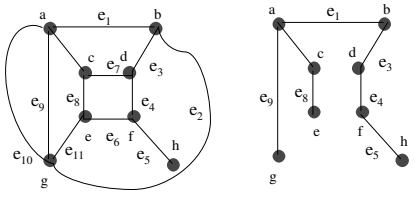
Spanning Trees

Aqui será discutido o problema de encontrar um sub-grafo *T* de um grafo *G* tal que *T é uma árvore que contém todos os vértices de G.*

A árvore *T* resultante é denominada de "spanning tree".

Em geral, um grafo pode conter várias "spanning trees".

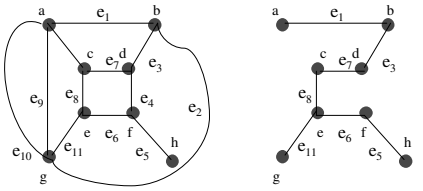
Exemplo de "spanning tree" (1)



Grafo

Spanning Tree

Exemplo de “spanning tree” (2)



Grafo

Spanning Tree

Teorema

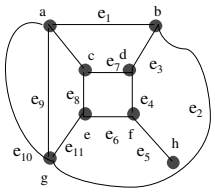
Um grafo *G* tem uma “spanning tree” se e somente se *G* é conectado.

Procurando uma “spanning tree” : busca em largura

Input : a connected graph *G* with vertices ordered $V = \{v_1, v_2, \ldots, v_n\}$
Output : a spanning tree *T*.
procedure *bfs(V, E)*
// *V'* := vertices of spanning tree *T* ; *E'* := edges of spanning tree *T* ;
// v_1 is the root of the spanning tree ; *S* is an ordered list ;
S := (v_1);
V' = $\{v_1\}$;
E' = $\varnothing$;
while *true* **do**
begin
for each $x \in S$, in order, **do**
for each $y \in V - V'$, in order, **do**
if *(x, y)* is an edge **then**
add edge *(x, y)* to *E'* and y to *V'*;
if no edges were added **then**
return *T* ;
S := children of *S* ordered consistently, with the original vertex ordering;
end;
end *bfs*.

Exercício - “breadth-first search”

Encontre uma “spanning tree” para o grafo *G* abaixo, utilizando o vértice *a* como raiz:



Solução (1)

- Inicialmente escolha uma ordem para os vértices de G : por exemplo, *abcdefgh.*
- Configuração inicial das principais variáveis:

  $E' = \varnothing \quad V' = \{a\} \qquad S = \{a\}$
- Iteração 1 :

  $E' = \{ (a,b), (a,c), (a,g) \}$

  $V' = \{ a, b, c, g \} \qquad S = \{b, c, g\}$
- Iteração 2 :

  $E' = \{ (a,b), (a,c), (a,g), (b,d), (c,e) \}$

  $V' = \{ a, b, c, d, e, g \} \qquad S = \{d, e\}$

Solução (2)

- Iteração 3 :

  $E' = \{ (a,b), (a,c), (a,g), (b,d), (c,e), (d,f) \}$

  $V' = \{ a, b, c, d, e, f, g \} \qquad S = \{ f \}$
- Iteração 4 :

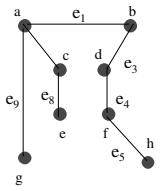
  $E' = \{ (a,b), (a,c), (a,g), (b,d), (c,e), (d,f), (f,h) \}$

  $V' = \{ a, b, c, d, e, f, g, h \} \quad S = \{ h \}$
- Iteração 5 :

  $E' = \{ (a,b), (a,c), (a,g), (b,d), (c,e), (d,f), (f,h) \}$

  $V' = \{ a, b, c, d, e, f, g, h \} \quad S = \varnothing$

Árvore resultante

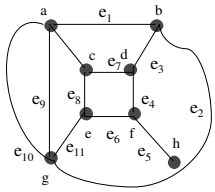


Procurando uma "spanning tree" : busca em profundidade

Input : a connected graph G with vertices ordered $V = \{v_1, v_2, \ldots, v_n\}$
Output : a spanning tree T.
procedure *dfs(V, E)*
// V' := vertices of spanning tree T ; E' := edges of spanning tree T ;
// v_1 is the root of the spanning tree ;
$V' := \{v_1\}$; $E' := \varnothing$; $w := v_1$;
while *true* **do**
begin
  while there is an edge *(w, v)* that when added to T does not create a cycle in T **do**
    begin
      choose the edge (w, v_k) with minimum k that when added to T does not create a cycle in T ;
      add (w, v_k) to E' ; add v_k to V';
      w := v_k ;
    end;
  if w = v_1 **then**
    return T ;
  w := parent of w in T ; // Backtrack.
end;
end *dfs*.

Exercício - “depth-first search”

Encontre uma “spanning tree” para o grafo *G* abaixo, utilizando o vértice *a* como raiz:



Solução (1)

- Inicialmente escolha uma ordem para os vértices de *G* : por exemplo, *abcdefgh.*
- Configuração inicial das principais variáveis:
  E’ = ∅ V’ = {a} w = a
- Passo 1 :
  E’ = { (a,b) }
  V’ = { a, b } w = b
- Passo 2 :
  E’ = { (a,b), (b,d) }
  V’ = { a, b, d } w = d

Solução (2)

- Passo 3 :
  E’ = { (a,b), (b,d), (d,c)}
  V’ = { a, b, c, d } w = c
- Passo 4 :
  E’ = { (a,b), (b,d),(d,c),(c,e) }
  V’ = { a, b, c, d, e } w = e

Solução (3)

- Passo 5 :
  E’ = { (a,b), (b,d), (d,c), (c,e), (e,f)}
  V’ = { a, b, c, d, e, f } w = f
- Passo 6 :
  E’ = { (a,b), (b,d),(d,c),(c,e), (e,f), (f,h) }
  V’ = { a, b, c, d, e, f, h } w = h // backtrack

Solução (4)

- Passo 7 :
  E' = { (a,b), (b,d), (d,c), (c,e), (e,f), (f,h) }
  V' = { a, b, c, d, e, f, h } w = f // backtrack
- Passo 8 :
  E' = { (a,b), (b,d), (d,c), (c,e), (e,f), (f,h) }
  V' = { a, b, c, d, e, f, h } w = e

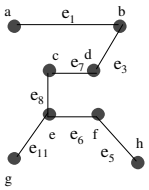
Solução (5)

- Passo 9 :
  E' = { (a,b), (b,d), (d,c), (c,e), (e,f), (f,h), (e,g) }
  V' = { a, b, c, d, e, f, h, g } w = e // backtrack
- Passo 10 :
  E' = { (a,b), (b,d), (d,c), (c,e), (e,f), (f,h), (e,g) }
  V' = { a, b, c, d, e, f, h, g } w = c // backtrack

Solução (6)

- Passo 11 :
  E' = { (a,b), (b,d), (d,c), (c,e), (e,f), (f,h), (e,g) }
  V' = { a, b, c, d, e, f, h, g } w = d // backtrack
- Passo 12 :
  E' = { (a,b), (b,d), (d,c), (c,e), (e,f), (f,h), (e,g) }
  V' = { a, b, c, d, e, f, h, g } w = b // backtrack
- Passo 13 :
  E' = { (a,b), (b,d), (d,c), (c,e), (e,f), (f,h), (e,g) }
  V' = { a, b, c, d, e, f, h, g } w = a

Árvore resultante

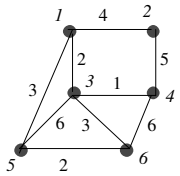


Comentário

- Ambos os métodos (busca em largura e busca em profundidade) podem ser utilizados para testar se um grafo *G* arbitrário é conectado.

Minimal Spanning Tree

O grafo ponderado *G* representa 6 cidades e os custos de construção de estradas que interligam duas cidades. Deseja-se construir um sistema de estradas que interligue todas as cidades e que seja o mais barato.



Esboço da solução

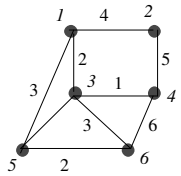
- A solução pode ser representada por um sub-grafo.
- O sub-grafo deve conter todos os vértices (cidades), deve ser conectado (qualquer cidade deve ser atingível a partir de uma determinada cidade) e deve ter um único caminho simples entre cada par de vértices (grafos contendo múltiplos caminhos simples não podem representar um sistema de custo mínimo).
- Assim, o sub-grafo deve ser uma "spanning tree", cuja soma dos pesos de suas arestas é minima. Esse tipo de árvore é denominado de "spanning tree" mínima.

Definição - Spanning Tree Mínima

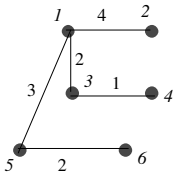
Seja *G* um grafo ponderado.

Uma "spanning tree" mínima de *G* é uma "spanning tree" de *G* com peso (somatória dos pesos das arestas de G) mínimo.

Exemplo - Spanning Tree Mínima



Grafo



Spanning tree mínima

Algoritmo de Prim

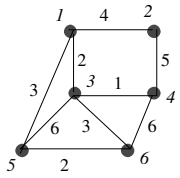
- Este algoritmo constrói uma árvore adicionando arestas iterativamente, até que uma "spanning tree" mínima seja gerada. A cada iteração é adicionada uma aresta de menor peso que não completa um ciclo para a árvore corrente.
- Entrada do algoritmo : um grafo ponderado conectado, com vértices 1, ..., *n* e vértice inicial *s*. Se *(i,j)* é uma aresta, *w(i,j)* é igual ao peso de *(i,j)*. Se *(i,j)* não é uma aresta, *w(i,j)* é igual a ∞ (um valor maior que qualquer peso existente).
- Saída do algoritmo : o conjunto de arestas *E* de uma "spanning tree" mínima.

Algoritmo de Prim

```
procedure prim(w, n, s)
// v(i) = 1 if vertex i has been added to minimal spanning tree (mst). v(i) = 0 otherwise.
for i := 1 to n do
  v(i) := 0 ;
v(s) := 1 ;  // add start vertex to mst.
E := ∅ ;  // begin with an empty edge set.
for i := 1 to n - 1 do  // put n - 1 edges in the minimal spanning tree.
  begin
    min := ∞ ;  // add edge of minimum weigth with one vertex in mst and one vertice not in mst.
    for j := 1 to n do
      if v(j) = 1 then  // j is a vertex in mst.
      for k = 1 to n do
        if v(k) = 0 and w(j,k) < min then
        begin
          add_vertex := k ;   e := (j,k) ;   min := w(j,k) ;
        end;
    v(add_vertex) := 1 ;  // put vertex and edge in mst.
    E := E ∪ {e} ;
  end ;
return E ;
end prim.
```


Exercício - algoritmo de Prim

Mostre como o algoritmo de Prim encontra uma "spanning tree" mínima para o grafo apresentado abaixo. Assuma que o vértice inicial é o vértice *1*.



Iteração 1

Aresta	Peso
(1,2)	4
(1,3)	2

	1	2	3	4	5	6
v(i)	1	0	0	0	0	0

	1	2	3	4	5	6
v(i)	1	0	1	0	0	0

E = { (1,3) }

Iteração 2

Aresta	Peso
(1,2)	4
(1,5)	3
(3,4)	1

	1	2	3	4	5	6
v(i)	1	0	1	0	0	0

	1	2	3	4	5	6
v(i)	1	0	1	1	0	0

E = { (1,3), (3,4) }

Iteração 3

Aresta	Peso
(1,2)	4
(1,5)	3

	1	2	3	4	5	6
v(i)	1	0	1	1	0	0

	1	2	3	4	5	6
v(i)	1	0	1	1	1	0

E = { (1,3), (3,4), (1,5) }

Iteração 4

Aresta	Peso
(1,2)	4
(3,6)	3
(5,6)	2

	1	2	3	4	5	6
v(i)	1	0	1	1	1	0

	1	2	3	4	5	6
v(i)	1	0	1	1	1	1

E = { (1,3), (3,4), (1,5), (5,6) }

Iteração 5

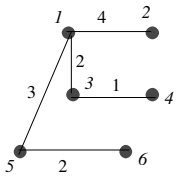
Aresta	Peso
(1,2)	4

	1	2	3	4	5	6
v(i)	1	0	1	1	1	1

	1	2	3	4	5	6
v(i)	1	1	1	1	1	1

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

Árvore resultante



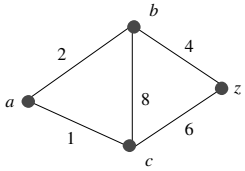
Teorema

O algoritmo de Prim é correto, isto é, ao término da execução do algoritmo, T é uma "spanning tree" mínima.

O algoritmo de Prim é um exemplo de um algoritmo "greedy", ou seja, um algoritmo que otimiza a escolha a cada iteração sem levar em consideração as escolhas prévias.

Algoritmo Greedy

A figura abaixo mostra o fato de que a seleção de uma aresta incidente sobre o vértice adicionado mais recentemente e que tenha peso mínimo não necessariamente produz o caminho mínimo.

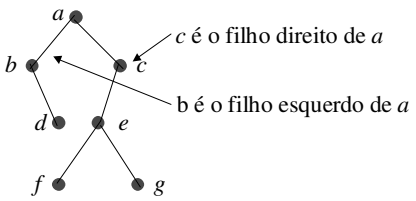


Começando a partir do vértice *a*, obtém-se (*a, c, z*), mas o caminho mínimo de *a* a *z* é (*a, b, z*).

Definição : árvores binárias

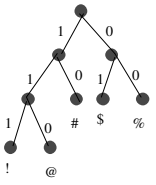
Uma árvore binária é uma árvore com raiz na qual cada vértice possui um filho, dois filhos ou nenhum filho. Além disso, cada filho é denominado filho direito (right child) ou filho esquerdo (left child). Quando um vértice possui dois filhos, um deles é denominado de filho direito e o outro recebe o nome de filho esquerdo.

Exemplo de árvore binária (1)



Exemplo de árvore binária (2)

Uma árvore que define um código de Huffman é uma árvore binária.



Teorema

Uma árvore binária completa é uma árvore binária cujos vértices possuem dois filhos ou nenhum filho. O teorema a seguir apresenta um resultado fundamental sobre árvores binárias completas.

Se T é uma árvore binária completa, com i vértices internos, então T possui $i+1$ vértices terminais e um total de $2i+1$ vértices.

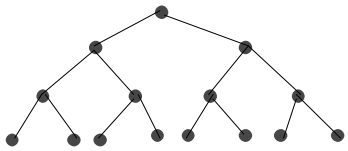
Teorema

Se uma árvore binária de altura h possui t vértices terminais, então

$$\log_2 t \leq h$$

Exemplo

Árvore binária de altura $h = 3$ e quantidade de vértices terminais $t = 8$. Para esta árvore binária $\log_2 t = h$.

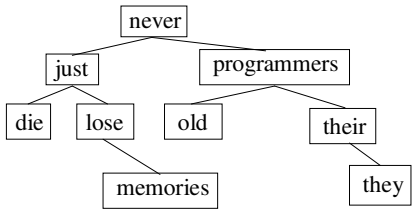


Definição : árvore de busca binária

Uma árvore de busca binária é uma árvore T na qual dados são associados aos vértices. Os dados são arranjados de tal forma que, para cada vértice v em T, cada item de dado na sub-árvore à esquerda de v é menor que o item de dado em v e cada item de dado na sub-árvore à direita de v é maior do que o item de dado em v.

Exemplo : árvore de busca binária

As palavras "old, programmers, never, die, they, just, lose, their, memories" podem ser colocadas numa árvore de busca binária da seguinte forma:



Algoritmo para construção de uma árvore de busca binária (1)

Entrada: uma sequência $w_1, \ldots, w_n$, de palavras distintas e o comprimento n da sequência.
Saída: uma árvore de busca binária T.

```
procedure make_bin_search_tree(w, n)
  let T be the tree with one vertex, root ;
  store w_1 in root ;
  for i := 2 to n do
  begin
    v := root ;
    search := true ;  // find spot for w_i
    while search do
      create binary search tree (see next slide) ;
    end ;
  end;
  return T ;
end make_bin_search_tree.
```


Algoritmo para construção de uma árvore de busca binária (2)

```
begin
  s := word in v ;
  if w_i < s then
    if v has no left child then
    begin
      add a left child l to v ;
      store w_i in l ;
      search := false ;  // end search
    end
    else
      v := left child of v ;
```

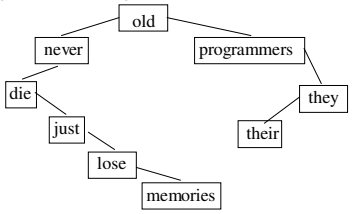
```
else
    if v has no right child then
    begin
      add a right child r to v ;
      store w_i in r ;
      search := false ;  // end search
    end
    else
      v := right child of v ;
end;
```


Exemplo de execução do algoritmo

Considere a seguinte sequência de palavras:

old programmers never die they just lose their memories

O algoritmo constrói a seguinte árvore de busca binária:



Comentários : árvores de busca binária

- Em geral existem <u>várias maneiras de se inserir dados</u> numa árvore de busca binária. Por exemplo, foram apresentadas nos exemplos anteriores, duas árvores de busca binária distintas para a mesma sequência de palavras.
- Árvore de busca binária são úteis para <u>localizar dados</u>.
- O tempo gasto na busca por um item de dado é maior quando o item não está contido na árvore de busca binária. Nesse caso, o procedimento de busca percorre o caminho mais longo, iniciando-se pela raiz. Assim o tempo máximo de busca por um item de dado é <u>aproximadamente proporcional à altura da árvore</u>. Assim, se a altura de uma árvore de busca binária é pequena, a pesquisa pela árvore sempre será rápida.

Métodos de percurso de árvores

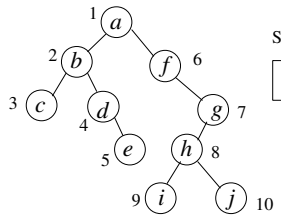
- A busca em largura e a busca em profundidade proporcionam métodos para "caminhar" numa árvore, isto é, percorrer a árvore sistematicamente de forma que cada vértice é visitado exatamente uma vez.
- A seguir serão considerados três métodos de percurso para árvores binárias: "preorder", "inorder" e "postorder".

Percurso "preorder"

Input : *PT*, the root of a binary tree
Output : dependent on how process command is interpreted.
If process is equivalent to print, then the output is the vertex visiting sequence.

```
procedure preorder(PT)
  if PT is empty then
     return ;
  process PT ;
  l := left child of PT ;
  preorder(l) ;
  r := right child of PT ;
  preorder(r) ;
end preorder.
```


Exemplo de percurso "preorder"



Sequência a ser impressa:

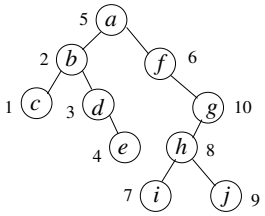
a b c d e f g h i j

Percurso "inorder"

Input : *PT*, the root of a binary tree
Output : dependent on how process command is interpreted.
If process is equivalent to print, then the output is the vertex visiting sequence.

```
procedure inorder(PT)
  if PT is empty then
     return ;
  l := left child of PT ;
  inorder(l) ;
  process PT ;
  r := right child of PT ;
  inorder(r) ;
end inorder.
```

Exemplo de percurso “inorder”



Sequência a ser impressa:

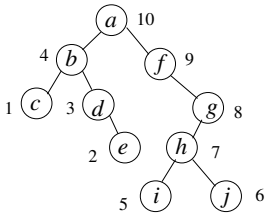
c b d e a f i h j g

Percurso “postorder”

Input : *PT*, the root of a binary tree
Output : dependent on how process command is interpreted.
If process is equivalent to print, then the output is the vertex visiting sequence.

```
procedure postorder(PT)
  if PT is empty then
     return ;
  l := left child of PT ;
  postorder(l) ;
  r := right child of PT ;
  postorder(r) ;
  process PT ;
end postorder.
```


Exemplo de percurso “postorder”



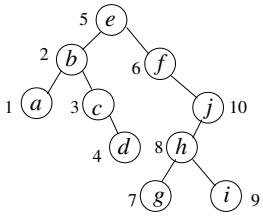
Sequência a ser impressa:

c e d b i j h g f a

Comentário - percurso de árvore

- Se os dados são armazenados numa árvore de busca binária, o percurso “inorder” irá processar os dados “em ordem”, uma vez que a sequência filho esquerdo/raiz/filho direito coincide com a ordenação dos dados na árvore.

Exemplo - percurso “inorder” numa árvore de busca binária

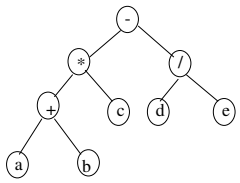


Sequência a ser impressa:

a b c d e f g h i j

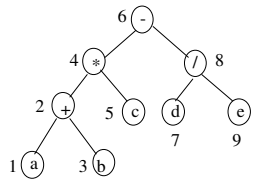
Exemplo - representação de expressões aritméticas (1)

Uma expressão do tipo *(a + b) * c - d / e* pode ser representada na forma de árvore binária. Os vértices terminais correspondem aos operandos e os vértices internos correspondem aos operadores.



Exemplo - representação de expressões aritméticas (2)

Se for feito o percurso “inorder” da árvore binária e inserido um par de parênteses para cada operação, obtém-se a chamada notação infixa:

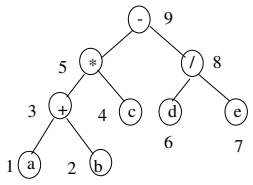


Sequência a ser impressa:

(((a+b) * c) - (d / e))

Exemplo - representação de expressões aritméticas (3)

Se for feito o percurso “postorder” da árvore binária, obtém-se a chamada expressão em notação polonesa reversa, ou notação pósfixa :

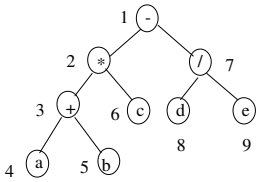


Sequência a ser impressa:

a b + c * d e / -

Exemplo - representação de expressões aritméticas (4)

Se for feito o percurso "preorder" da árvore binária, obtém-se a chamada expressão em notação polonesa, ou notação prefixa :

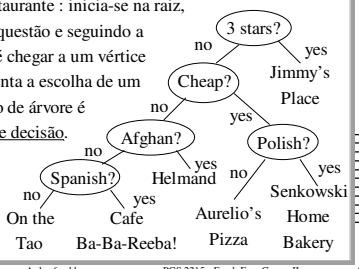


Sequência a ser impressa:

- * + a b c / d e

Árvores de decisão

A árvore binária ao lado fornece um procedimento para escolher um restaurante : inicia-se na raiz, respondendo a cada questão e seguindo a aresta apropriada, até chegar a um vértice terminal, que representa a escolha de um restaurante. Esse tipo de árvore é chamada de árvore de decisão.

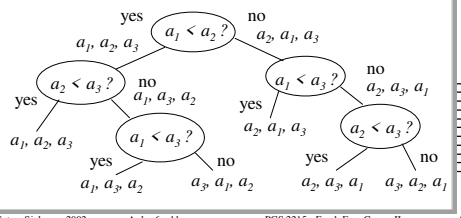


Tempo mínimo para ordenação (1)

- Pode-se utilizar árvores de decisão para estimar o caso de pior tempo (worst-case time) para ordenação.
- Problema de ordenação : dado n itens x_1 , ..., x_n , colocá-los em ordem crescente (ou decrescente).
- A análise será restrita aos algoritmos de ordenação que comparam repetidamente dois elementos e que, com base no resultado da comparação, modificam a lista original.

Tempo mínimo para ordenação (2)

Um algoritmo para ordenar a_1, a_2, a_3, é dado pela árvore de decisão abaixo. Cada aresta é rotulada com o arranjo da lista com base na resposta da pergunta do vértice interno. Os vértices terminais apresentam a lista ordenada.



Tempo mínimo para ordenação (3)

Definindo o tempo de pior caso para ordenação como sendo o número de comparações no pior caso, segue que a altura da árvore de decisão que resolve o problema de ordenação é igual ao tempo de pior caso.

Demonstra-se que esse algoritmo é o caso ótimo, ou seja, nenhum algoritmo que ordena três itens tem uma performance melhor que 3 comparações.

Teorema

Se $f(n)$ é o número de comparações necessárias para ordenar n itens no pior caso por um algoritmo de ordenação, então

$$f(n) = \Omega(n \log_2 n)$$

Bibliografia

[1] Johnsonbaugh, R. Discrete Mathematics. Prentice Hall International, London, UK, 4th. Ed. 1997. Cap. 7.
[2] Cormem, T. H., Leiserson, C. E., and Rivest, R. L. Introduction to Algorithms, MIT Press, Cambridge, Mass., 1990.