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;*

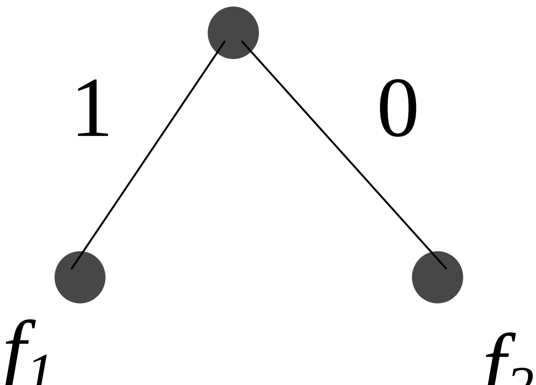

Figure 7.1.9

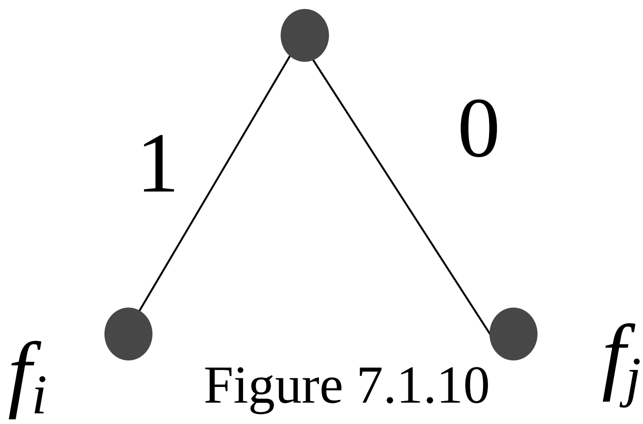

Figure 7.1.10

Procurando uma “spanning tree” : <u>busca em largura</u>

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*.

Procurando uma “spanning tree” : <u>busca em profundidade</u>

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*.

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.
```

Algoritmo para construção de uma árvore de busca binária

Entrada: uma sequência w_1 , ... , 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
      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;
  return T ;
end make_bin_search_tree.
```

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*.

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*.

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*.