Programa 7.19 *Implementação do algoritmo de Prim para obter a árvore geradora mínima*

```
procedure AgmPrim (var Grafo: TipoGrafo; var Raiz: TipoValorVertice);
var Antecessor: array[TipoValorVertice] of integer;
    P         : array[TipoValorVertice] of TipoPeso;
    Itensheap : array[TipoValorVertice] of boolean;
    Pos       : array[TipoValorVertice] of TipoValorVertice;
    A         : TipoVetor;
    u, v      : TipovalorVertice;

{-- Entram aqui os operadores do tipo grafo do Programa 7.3   --}
{-- ou do Programa 7.5 ou do Programa 7.7, e os operadores    --}
{-- RefazInd, RetiraMinInd e DiminuiChaveInd do Programa 7.18 --}

begin { AgmPrim }
  for u := 0 to Grafo.NumVertices do
    begin {Constroi o heap com todos os valores igual a INFINITO}
    Antecessor[u] := -1;
    p[u] := INFINITO;
    A[u+1].Chave := u; {Heap a ser construido}
    ItensHeap[u] := true;
    Pos[u] := u+1;
    end;
  n := Grafo.NumVertices;
  p[Raiz] := 0;
  Constroi (A);
  while n >= 1 do {enquanto heap nao estiver vazio}
    begin
    u := RetiraMinInd(A).Chave;
    ItensHeap[u] := false;
    if (u <> Raiz)
    then write ('Aresta de arvore: v[',u,'] v[',Antecessor[u],']');
    readln;
    if not ListaAdjVazia (u,Grafo)
      then begin
           Aux := PrimeiroListaAdj (u,Grafo);
           FimListaAdj := false;
           while not FimListaAdj do
             begin
             ProxAdj (u, Grafo, v, Peso, Aux, FimListaAdj);
             if ItensHeap[v] and (Peso < p[v])
             then begin
                  Antecessor[v] := u;
                  DiminuiChaveInd (Pos[v],Peso,A);
                  end
             end;
           end;
    end;
end; { AgmPrim }
```


Análise O desempenho do algoritmo de Prim depende da forma como a **fila de prioridades** é implementada. Se a fila de prioridades é implementada como um *heap* (veja Seção 4.1.5), podemos usar o procedimento Constroi do Programa 4.10 para a inicialização de A no Programa 7.19. O corpo do anel **while** é executado $|V|$ vezes, e, desde que o procedimento Refaz tem custo $O(\log |V|)$, o tempo total para executar a operação retira o item com menor peso é $O(|V| \log |V|)$. O **while** mais interno para percorrer a lista de adjacentes é executado $O(|A|)$ vezes ao todo, uma vez que a soma dos comprimentos de todas as listas de adjacência é $2|A|$. Dentro desse anel, o teste para verificar se o vértice v pertence ao *heap* A tem custo $O(1)$ pelo fato de o teste ser implementado mediante uma consulta a um arranjo de *bits*. O arranjo ItensHeap de *bits* é atualizado quando o vértice é retirado do *heap* (*ItensHeap*$[v]$ é tornado *false*). Após testar se v pertence ao *heap* A e o peso da aresta (u, v) é menor do que $p[v]$, o antecessor de v é armazenado em Antecessor e uma operação DiminuiChave é realizada sobre o *heap* A na posição $Pos[v]$, a qual tem custo $O(\log |V|)$. Logo, o tempo total para executar o algoritmo de Prim é $O(|V| \log |V| + |A| \log |V|) = O(|A| \log |V|)$.

7.8.3 Algoritmo de Kruskal

Assim como o algoritmo de Prim, o algoritmo de Kruskal para obter uma árvore geradora mínima pode ser derivado do algoritmo genérico apresentado no Programa 7.17. No algoritmo de Kruskal, o conjunto S é uma floresta e a aresta segura adicionada a S é sempre uma aresta de menor peso que conecta dois componentes distintos. A Figura 7.19 ilustra a execução do algoritmo de Kruskal sobre o grafo da Figura 7.16(a). Arestas em negrito pertencem à floresta sendo construída.

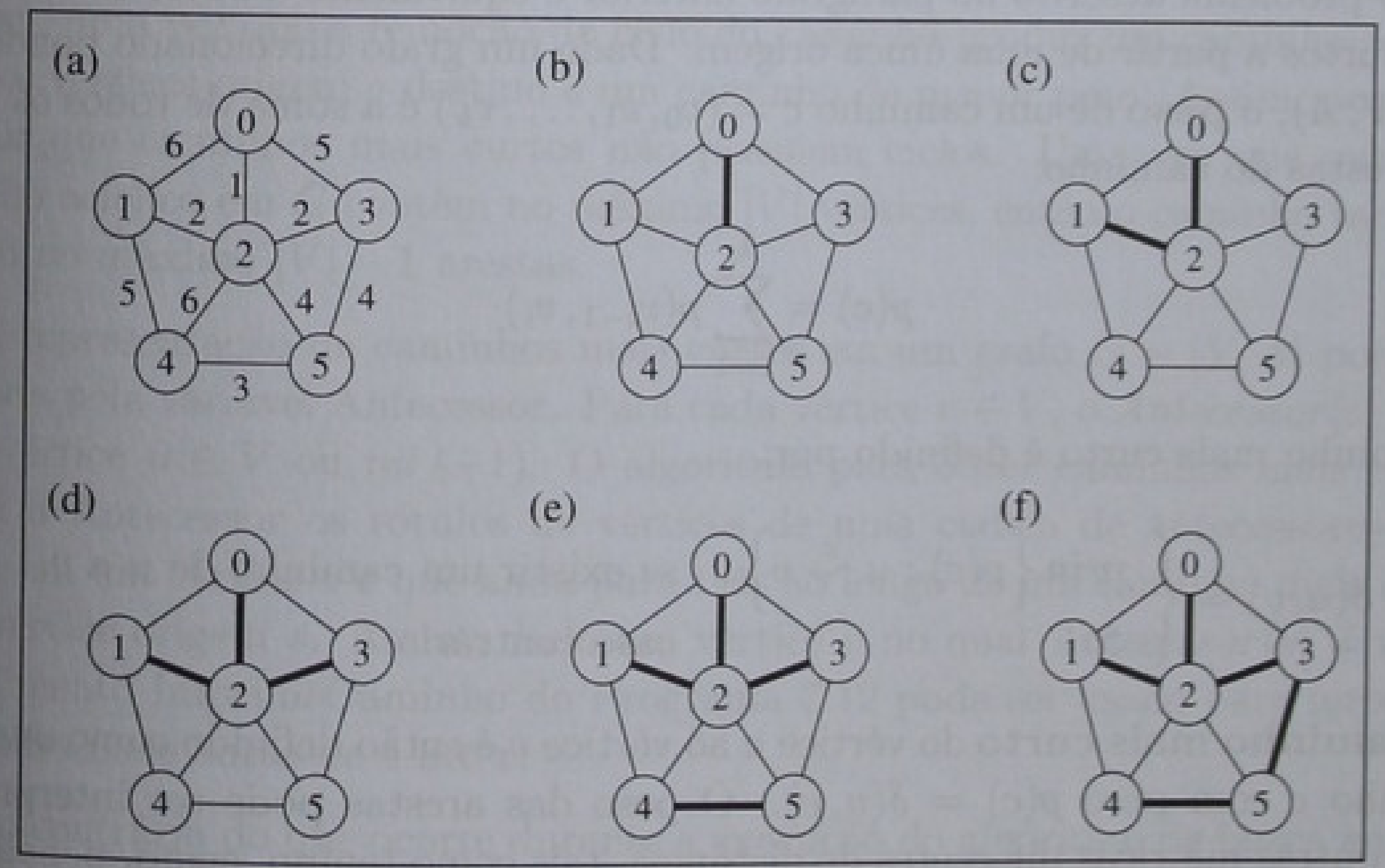

Figura 7.19 *Execução do algoritmo de Kruskal sobre o grafo da Figura 7.16(a).*