MO417 – Ata do Exercício 16.1-3

Ewerton Almeida Silva

5 de maio de 2010

Enunciado: Vamos supor que temos um conjunto de atividades para programar entre um grande número de salas de conferências. Desejamos programar todas as atividades usando o mínimo possível de salas de conferências. Forneça um algoritmo guloso eficiente para determinar que atividade deve usar cada sala de conferências.

(Isso também é conhecido como o ***problema de colorir grafos de intervalos***. Podemos criar um grafo de intervalos cujos vértices sejam as atividades dadas e cujas arestas conectem atividades incompatíveis. O menor número de cores necessárias para colorir cada vértice de tal modo que dois vértices adjacentes nunca recebam a mesma cor corresponde a encontrar o menor número de salas de conferências necessárias para programar todas as atividades dadas.)

Respostas: Dois raciocínios diferentes foram descritos em sala como soluções para este exercício. A seguir, são mostrados esses raciocínios a partir dos respectivos algoritmos e de uma breve explicação de cada um.

Solução de Marcos Cirne: A idéia do algoritmo proposto consiste em se armazenar em uma lista (encadeada) de salas, um conjunto compatível de atividades. Esse conjunto é encontrado utilizando-se o algoriitmo Greedy-Activity-Selector. Assim, este algoritmo tem tempo de execução $O(n^2)$.

Observação: Os parâmetros **s** e **f** indicam, respectivamente, as sequências de tempos de início e fim das atividades. É necessário que as atividades sejam organizadas por tempo de término monotonicamente crescente.

Algoritmo 1 LECTURE-HALL(s, f)

```
1: n ← length[s]
2: for i ← 1 to n do
3:       S[i] ← nil {S é um vetor de listas de atividades}
4: end for
5: k ← 1 {contador de atividades}
6: while s ≠ ∅ do
7:       S[k] ← GREEDY-ACTIVITY-SELECTOR(s,f)
8:       {Remover de s e f as atividades em S[k]}
9:       k ← k − 1
10: end while
11: return S
```

Algoritmo 2 GREEDY-ACTIVITY-SELECTOR(s, f)

```
1: n ← length[s]
2: A ← {1}
3: i ← 1
4: for m ← 2 to n do
5:       if s_m ⩾ f_i then
6:             A ← A ∪ {a_m}
7:             i ← m
8:       end if
9: end for
10: return A
```

Solução de Alexandre Passos: O algoritmo utiliza uma árvore de busca binária **aproximadamente balanceada** (por exemplo, uma árvore rubro-negra), na qual cada nó representa uma sala de conferência. Mais especificamente, um nó é um registro com os campos chave e uma lista de atividades. A chave de um nó contém o tempo de término da atividade que é concluída mais tarde, enquanto a lista guarda quais atividades já foram programadas para a sala (nó). Assim, para cada atividade, percorremos a árvore comparando o tempo de término da atividade atual com a chave do nó. Porque a operação de busca nessa árvore leva tempo $O(lgn)$ e há n atividades a serem programadas, a complexidade do algoritmo será $O(nlgn)$.

Observações:

1 . Os parâmetros **s** e **f** indicam, respectivamente, as sequências de tempos de início e fim das atividades. É necessário que as atividades sejam

organizadas por tempo de término monotonicamente crescente.

2. As operações de Inserção e Exclusão de um nó na árvore estão abstraídas no algoritmo. A Exclusão é usada no momento em que uma nova atividade é inserida na lista de um nó: o nó é excluído da estrutura, a lista do mesmo é incrementada com a nova atividade e sua chave recebe o tempo de término dessa atividade. Depois, o nó (sala) é novamente inserido na lista. Ainda, a inserção é usada para incluir uma nova sala na árvore.

Algoritmo 3 LECTURE-HALL(s, f)

```
1: Coloque num vetor A as atividades ordenadas pelo tempo de término
2: Salas ← arvorevazia
3: for i ← 1 to n {n é o número de atividades} do
4:     Buscar o nó v em Salas com maior chave t tal que t ⩽ s[A[i]]
5:     if existir tal v then
6:         Inclua A[i] em v.L {L é uma lista de atividades}
7:     else
8:         Crie um novo nó v
9:         Coloque A[i] em v
10:        Insira v em Salas
11:    end if
12: end for
13: return Salas
```