MAC 338 - Análise de Algoritmos

Departamento de Ciência da Computação

Primeiro semestre de 2003

Algoritmos gulosos

Considere um conjunto $S = \{1, 2, \ldots, n\}$ de n atividades que competem por um recurso. Por exemplo, o recurso poderia ser uma sala de aula, que comporta a cada momento apenas uma atividade. Suponha que cada atividade tem um tempo de início s_i e um tempo de término t_i, onde $0 \leq s_i \leq t_i$. Mais exatamente, entenda que o intervalo requerido pela atividade i é $[s_i, t_i)$. Duas atividades i e j são *compatíveis* se os intervalos $[s_i, t_i)$ e $[s_j, t_j)$ não se interseptam, ou seja, se $s_i \geq t_j$ ou $s_j \geq t_i$.

O problema da escolha de atividades (também conhecido como *problema da coleção máxima de intervalos disjuntos*) consiste no seguinte: encontrar o conjunto de atividades mutuamente compatíveis de tamanho máximo.

Consideremos que as atividades estão ordenadas pelo tempo de término, ou seja, que $t_1 \leq t_2 \leq \ldots \leq t_n$. (Caso isso não seja verdade, ordene-as em tempo $O(n \log n)$.)

É possível projetar um algoritmo de programação dinâmica para esse problema. Para tanto, faça $t_0 := 0$, $s_{n+1} := \infty$ e denote por $c[i, j]$ o tamanho de uma solução do problema para o conjunto $S_{ij} := \{k : s_k \geq t_i \text{ e } t_k \leq s_j\}$. Observe que $c[0, n+1]$ é exatamente o tamanho de uma solução do problema para S.

O número $c[i, j]$ pode ser calculado por meio da seguinte recorrência:

$$c[i,j] = \begin{cases} 0 & \text{se } S_{ij} = \emptyset, \\ \max_{k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{se } S_{ij} \neq \emptyset. \end{cases}$$

Para compreender a recorrência acima, considere uma solução R do problema para S_{ij}. Por definição, temos que $R \subseteq S_{ij}$ e $c[i, j] = |R|$. Assim, é evidente que $c[i, j] = 0$ se $S_{ij} = \emptyset$. Por outro lado, se $S_{ij} \neq \emptyset$, então R contém pelo menos uma atividade de S_{ij}. Seja k um elemento de R. Temos que R pode ser decomposto como $R_1 \cup \{k\} \cup R_2$, onde $R_1 \subseteq S_{ik}$ e $R_2 \subseteq S_{kj}$. Mas, como R tem tamanho máximo, R_1 e R_2 têm que ser soluções do problema para S_{ik} e S_{kj} respectivamente (exercício 1). Daí segue que $c[i, j] = c[i, k] + 1 + c[k, j]$. Como não temos em mãos uma solução do problema para S_{ij} (ou seria trivial calcular $c[i, j]$), temos que testar todos os candidatos a k, ou seja, todos os elementos de $S_{i,j}$ e escolher um que resulte no maior valor para $c[i, k] + c[k, j] + 1$. Tal k faz parte de uma solução do problema para S_{ij}. A recorrência reflete esse fato.

Segue um algoritmo de programação dinâmica baseado na recorrência acima. O algoritmo calcula os valores de $c[i, j]$ iterativamente.

```
seleciona_atividades_PD (s,t,n) {
  para i := 0 até n+1 faça c[i,j] := 0;
  para l := 1 até n+1 faça
    para i := 0 até n+1-l faça {
      j := i+l;
      max := infinito;
      para k := i+1 até j-1 faça
        se (s[k] >= t[i] e t[k] <= s[j]) {
          tam := c[i,k]+c[k,j]+1;
          se (tam > max) max := tam;
        }
    }
  devolva c[0,n+1];
}
```

Claramente o algoritmo consome tempo $\Theta(n^3)$. Ele na verdade não resolve o problema, mas apenas calcula o tamanho de uma solução do problema. É fácil, no entanto, modificá-lo para que construa também uma solução (exercício 2). Basta utilizar uma outra matriz para guardar, para cada $c[i,j]$ com $j-i>0$, um valor de k para o qual $c[i,j]=c[i,k]+c[k,j]+1$, ou seja, um k em que o máximo é atingido na recorrência. Depois, pode-se utilizar essa segunda matriz facilmente para construir uma solução ótima. No total, o algoritmo completo consome ainda $\Theta(n^3)$.

Há, no entanto, um algoritmo mais simples e eficiente para o problema! É um algoritmo *guloso*. Tal algoritmo parte de A vazio e, repetidamente, inclui em A a próxima atividade que é compatível com as que já foram incluídas em A. Surpreendentemente, o conjunto A ao final é uma solução do problema! O algoritmo resultante segue.

```
seleciona_atividades_guloso (s,t,n) {
  A := {1};
  ult := 1;
  para i := 2 até n faça
    se (s[i] >= t[j]) {
      A := A U {i};
      ult := i;
    }
  devolva A;
}
```

A complexidade de tal algoritmo é $O(n)$ e ele produz uma solução do problema para S, como passamos a demonstrar.

Seja R uma solução do problema para S tal que $R \cap A$ é máximo. Digamos que $R=\{y_1,\ldots,y_m\}$, com $y_1>\ldots>y_m$, e digamos que $A=\{x_1,\ldots,x_t\}$, com $x_1>\ldots>x_t$ e, claro, $t \le m$. Se $R \subseteq A$, então $A=R$ e não há nada a provar. Então suponhamos que $R \not\subseteq A$. Seja i o menor índice tal que $y_i \notin A$. Se $i>t$, então $A \subseteq R$, $y_i > y_t = x_t$ e y_i é compatível com todas as atividades em A, uma contradição, pois isso implicaria que o algoritmo guloso acima teria adicionado mais algum elemento a A. Se, por outro lado, $i \le t$, então $y_i > x_i$ (pela escolha gulosa de x_i), que implica que $t_{y_i} \ge t_{x_i}$. Além disso, $s_{x_i} \ge t_{y_{i-1}}$, pois x_i é compatível com $\{x_1,\ldots,x_{i-1}\}=\{y_1,\ldots,y_{i-1}\}$. Ou seja, podemos trocar x_i por y_i em R, obtendo R', que é também uma coleção de atividades mutuamente compatíveis e tem o mesmo tamanho que R. Ou seja, R' é também uma solução do problema para S. Porém $|R' \cap A| > |R \cap A|$, contrariando a escolha de R. Isso completa a prova que A é uma solução do problema para S.

Exercícios

1. Mostre que, se R_1 não é uma solução do problema para S_{ik}, então R não é uma solução do problema para S_{ij}. Faça o mesmo para R_2 com S_{kj} no lugar de S_{ik}.

2. Modifique o algoritmo `seleciona_atividades_PD` para que este construa uma solução do problema, não apenas calcule o seu tamanho. Seu algoritmo deve consumir tempo $O(n^3)$.