UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
ESCOLA DE INFORMÁTICA APLICADA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Terceiro Trabalho
Algoritmos Gulosos - Seleção de Atividade

Bruno Lírio Alves
Pedro Paulo Gouveia
Thales Veras

Vânia Felix

Rio de Janeiro, 31 de outubro de 2013.

2. Motivação

2.1 Definição do Problema

O problema da seleção de atividades, dado um conjunto de atividades, definidas por seus tempos de início e fim, alocar o maior número possível de atividades em um período de tempo definido, por exemplo, um conjunto de palestras ou de sala de aula que devem ser alocadas em um auditório ou em uma aula.

Consiste em constuir a solução aos poucos, o candidato é escolhido e adicionado à solução ele permanece para sempre mas sem nunca voltar atrás das suas atividades, uma vez que os algoritmos gulosos definem que um elemento está na solução do problema, uma vez que um candidato é excluído do conjunto solução, ele nunca mais é reconsiderado, este elemento nunca será retirado jamais. Procede acrescentando os n
ovos elementos, um de cada vez.

Mais exemplos da seleção de atividades:

- Existem diversas atividades (aulas) que querem usar um mesmo recurso (uma sala de aula).
- Cada atividade tem um horário de início e um horário de fim.
- Só existe uma sala disponível.
- Duas salas não podem ser ministradas na mesma sala ao mesmo tempo, ou seja, conflito de horário.

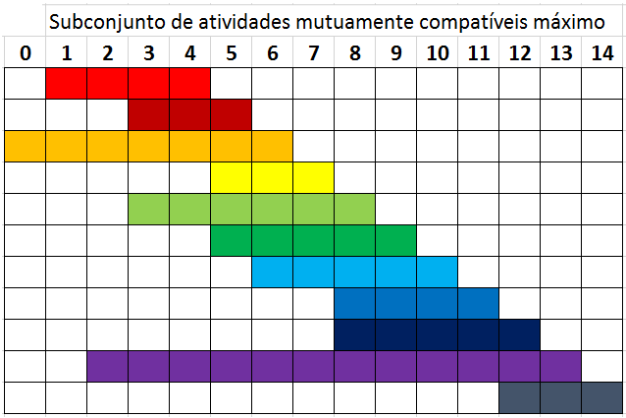
Definição Formal do Problema (Entrada / Questão / Saída)

- Entrada: Um algoritmo guloso seleciona, a cada passo, o melhor elemento.
- Questão: Verifica se ele é viável, vindo a fazer parte da solução ou não.
- Saída: Após uma seqüência de decisões, a solução do problema é alcançada.

Na seqüência de decisões, nenhum elemento é examinado mais de uma vez: ou ele fará parte da saída, ou será descartado.

2.2. Exemplos de Instâncias do Problema

Conjunto de Atividades											
i	1	2	3	4	5	6	7	8	9	10	11
s[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14

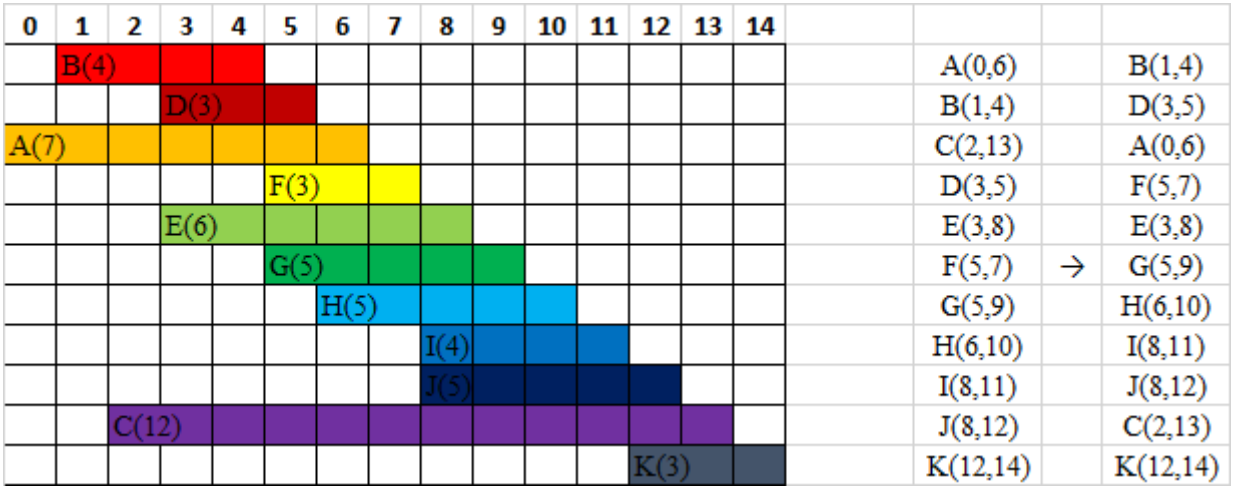


Dado um conjunto S com n atividades, onde s_i e f_i são respectivamente o tempo de início e de fim da atividade i, 1 <= i <= n, onde n são aulas. As duas atividades são compatíveis entre si quando seus tempos de início e de fim não se sobrepõem. O seu objetivo é encontrar um subconjunto máximo de A com atividades compatíveis.

3. Algoritmo

3.1. Descrição da ideia do algoritmo (como ele resolve o problema)

Assumir que as atividades estão ordenadas de forma crescente no tempo de finalização.



Dada uma coleção de atividades, digamos S, dadas em intervalos, determinar um subconjunto sem sobreposição (compatíveis) máximo de atividades de S, por exemplo da imagem acima, são 11 atividades, em 14 unidades de tempo.

3.2. A partir da ideia, mostrar exemplos* de solução para as instâncias apresentadas

O problema de Seleção de atividades consiste em agendar a utilização de um determinado recurso por um conjunto de atividades. Suponhamos que tenhamos um conjunto S = {a1, a2, .., an} de n atividades, onde a_1, a_2,...,a_n são cada aula, as propostas que desejam utilizar um mesmo recurso, recurso tal como uma sala de aula, que pode ser utilizada por apenas uma atividade por um determinado período de tempo. Cada atividade tem um tempo de início s_i e um tempo de término f_i, onde si < fi. Duas atividades a_i e a_j são compatíveis se os intervalos [s_i, f_i) e [s_j , f_j) não se sobrepõem, i.e., se s_i >= f_j ou se s_j >= f_i. O problema de Seleção de Atividades consiste em encontrar o subconjunto de tamanho máximo de atividades mutuamente compatíveis.

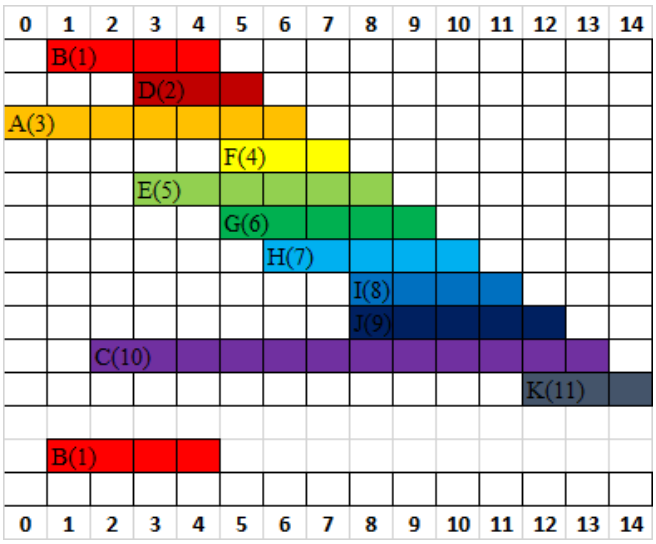
3.3. Descrição Formal do Algoritmo

```
SeleçãoAtividades(s, f)
n ← length(s)
S ←{1}
j ←1
for i ←2 to n do
      if s_i >= f_j then {
            S ← S ∪{i}
            j ← i
      }
Return S
```

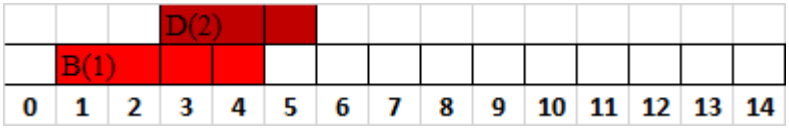
3.4. Etapas da aplicação da técnica (*Exemplo de solução)

As 11 atividades seguintes:
ínicio: s[i] = {1,3,0,5,3,5, 6, 8, 8, 2, 12}
fim: f[i] = {4,5,6,7,8,9,10,11,12,13,14}
O conjunto máximo de atividades que podem ser executadas:
{0,3,7,10}

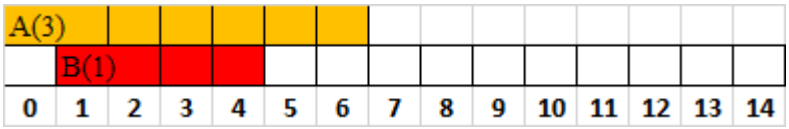
Primeira Etapa:
Ínicio: conjunto solução S recebe primeira aula, aula 1 (B)



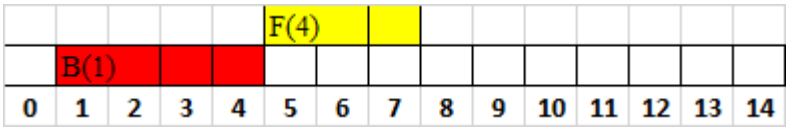
Segunda Etapa:
Primeira iteração: Testa se início de aula 2 (D) >= final de aula 1 (B)



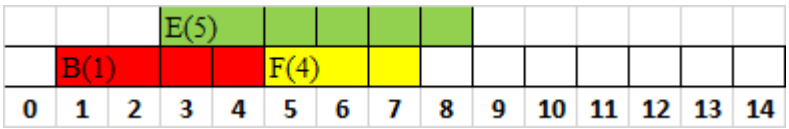
Terceira Etapa:
Testa início de A >= final de B



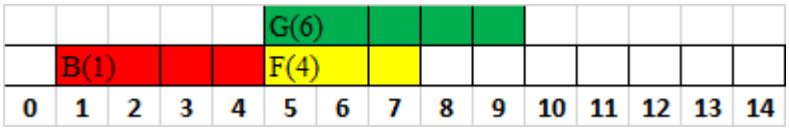
Quarta Etapa:
Testa início F >= final de B.
Ok: aula 4 (F) adiciona ao conjunto solução S



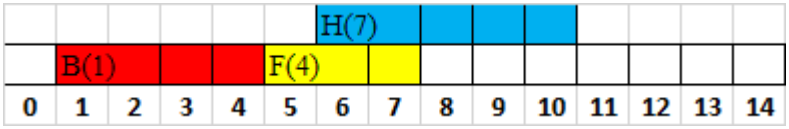
Quinta Etapa:
Testa início de E >= final de F



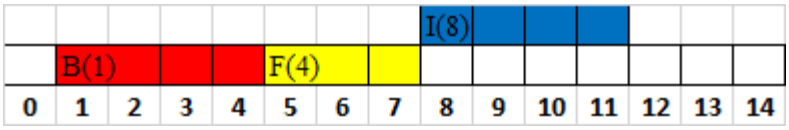
Sexta Etapa:
Testa início de G >= final de F



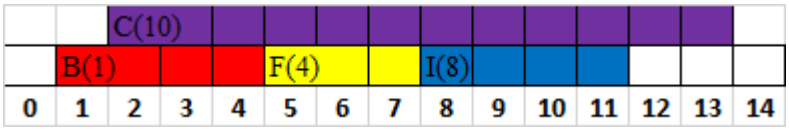
Sétima Etapa:
Testa início de H >= final de F



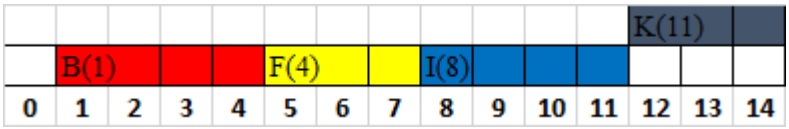
Oitava Etapa:
Testa início I >= final de F.
Ok: Adiciona I ao conjunto solução S.



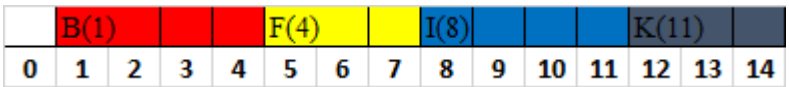
Nona Etapa:
Testa início C >= final de I.



Décima Etapa:
Testa início K >= final de I.
Ok: Adiciona K ao conjunto solução S.



Conjunto solução:
S = {B,F,I,K}



3.5. Opcional: caso queiram, após a descrição formal do algoritmo, aí sim apresenta-se uma implementação/codificada em linguagem de programação

```
#include<stdio.h>
// Prints a maximum set of activities that can be done by a single
// person, one at a time.
//  n   -->  Total number of activities
//  s[] -->  An array that contains start time of all activities
//  f[] -->  An array that contains finish time of all activities


void printMaxActivities(int s[], int f[], int n){
   int i, j;

   printf ("Following activities are selected \n");

   // The first activity always gets selected
   i = 0;
   printf("%d ", i);

   // Consider rest of the activities
   for (j = 1; j < n; j++)
   {
     // If this activity has start time greater than or equal to the finish
     // time of previously selected activity, then select it
     if (s[j] >= f[i])
     {
         printf ("%d ", j);
         i = j;
     }
   }
}


 // driver program to test above function
 int main()
 {
     int s[] =  {1,3,0,5,3,5,6, 8, 8, 2, 12};
     int f[] =  {4,5,6,7,8,9,10,11,12,13,14};
     int n = sizeof(s)/sizeof(s[0]);
     printMaxActivities(s, f, n);
     getchar();
```

```
    return 0;
}
```

```
Output:
Following activities are selected
0 3 7 10
```

4. Análise do Algoritmo

4.1. Prova de corretude (mostrar que o algoritmo funciona de fato)

Considerando a implementação:

```
SeleçãoAtividades(s, f)
n ← length(s)
A ←{1}
j ←1
for i ←2 to n do
     if s_i >= f_j then {
          A ← A ∪{i}
          j ← i
     }
Return A
```

e que o algoritmo retorna um conjunto na forma A = {1,4,5,...}.

E nos apoiando sobre o seguinte Teorema:

Considere qualquer subproblema não vazio de Sij, e seja a_m a atividade em Sij com o tempo de término mais antigo;

Então

1. A atividade a_m é usada em algum subconjunto de tamanho máximo de atividades mutuamente compatíveis de Sij.

2. O subproblema Sim é vazio, de forma que a escolha de a_m deixa um subproblema Smj como únicoque pode não ser vazio.

Segue a prova deste teorema (Cormen):

Provaremos a segunda parte primeiro, pois ela é um pouco mais simples. Suponha que Sim é não vazio, de forma que existe alguma atividade a_k tal que $f\_i \leq s\_k \leq f\_k \leq s\_m < f\_m$. Então, a_k também está em Sije tem um tempo de término anterior à de a_m, o que contradiz nossa escolha de a_m. Concluímos que Sim é vazio.

Para provar a primeira parte, supomos que Aij é um subconjunto de tamanho máximo de atividades mutuamente compatíveis de Sij, e vamos ordenar as atividades em Aij em ordem monotonicamente crescente de tempo de término. Seja a_k a primeira atividade em Aij. Se a_k = a_m, terminamos, pois mostramos que a_m é usada em algum subconjunto de tamanho máximo de atividades mutuamente compatíveis de Sij. Se a_k ≠ a_m, construímos o subconjunto A'ij = {a_k} ∪ {a_m}. As atividades em A'ij são disjuntas, pois as atividades em Aij o são, a_k é a primeira atividade em Aij a terminar e f_m ≤ f_k. Observado que A'ij tem o mesmo número de atividades de Aij, vemos que A'ij é um subconjunto de tamanho máximo de atividades mutuamente compatíveis de Sij que inclui a_m.

Assim, considerando os tempos de término ordenados monotonicamente e S ={1,2,3...n}:

$$f\_1 \leq f\_2 \leq \ldots \leq f\_n$$

f_1 é o menor tempo de término, *vamos mostrar* que existe uma solução ótima A tal que 1 ∈ A (Teorema). Seja C ⊆ S uma solução ótima. Vamos Supor que as atividades também estejam ordenadas em C. Seja k a 1ª atividade escolhida na solução C. Então, necessariamente f_1 ≤ f_k. Portanto, a solução A = C-{k} ∪{1} é viável e é maximal, pois |C| = |A| (ou seja, existe mais de uma conjunto ótimo que apesar de poderem possuir elementos diferentes, o número destes elementos é o mesmo) .

Uma vez que A é uma solução ótima para S, então A1 = A - {1} é uma solução ótima, o que decorre do exposto acima, para o problema gerado por S1 ={ i ∈ S | s_i ≥ f_1 } (Teorema). Se fosse possível conseguir uma solução B1 com mais atividades que A1, então seria possível conseguir uma solução B para S com mais atividades que A, o que seria uma contradição.

Então, após a escolha local ótima resta um subproblema dado por S1 = { i ∈ S| s_i ≥ t_1}, independente da primeira escolha, que deve conter na sua solução a atividade em S1 com o menor tempo de término mais antigo. Esta mecânica de escolha se propaga ao longo das decisões, justificando o uso do algoritmo.

4.2. Demonstração da complexidade/tempo do algoritmo (geralmente aborda-se o pior caso)

```
SeleçãoAtividades(s, f)
n ← length(s)
S ←{1}
j ←1
for i ← 2 to n do
      if s_i >= f_j then {
            S ← S ∪{i}
            j ← i
Return S
```

Pelo algoritmo podemos ver que o loop é o passo determinante em relação ao tempo, temos então que a complexidade de tempo do algoritmo é Θ(n). Contudo, devemos ordenar os tempos de término das atividades, o que consome o tempo de O(n log n). Finalmente podemos concluir que a complexidade de tempo do algoritmo é O(n log n).

5. Conclusão e Discussões

Conclusão: O algoritmo sempre nos dará uma solução ótima porém, não teremos como saber se existem outras.

5.1. Discutir as vantagens e desvantagens do procedimento adotado.

Vantagens:

- Algoritmo simples.
- Fácil de implementação.
- Geralmente é fácil de construir um algoritmo.
- Baixa de complexidade.

Desvantagens:

- Não há nenhuma garantia de qualidade da solução.

5.2. É possível ter um algoritmo melhor?

Se considerarmos que os tempos de término estão previamente ordenados, teremos um algoritmo com melhor, pois sua complexidade se reduz a Θ(n).

6. Referências Bibliográficas

Algoritmos gulosos: definições e aplicações (2004). UNICAMP. ***Site.*** Disponível em: <http://www.ic.unicamp.br/~rocha/msc/complex/algoritmosGulososFinal.pdf>. Acesso em: 30 out. 2013

Activity selection problem (2013). WIKIPEDIA. ***Site.*** Disponível em: <http://en.wikipedia.org/wiki/Activity_selection_problem>. Acesso em: 30 out. 2013

Greedy Algorithms Activity selection problem. GEEKSFORGEEKS. ***Site.*** Disponível em: <http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/>. Acesso em: 30 out. 2013

Cormen, Algoritmos Teoria e Prática 2ª ed. ELSEVIER.