Teoria dos Grafos
Aula 19

Aula passada

- Clusterização (ou agrupamento)
- Algoritmo
- Variação

Aula de hoje

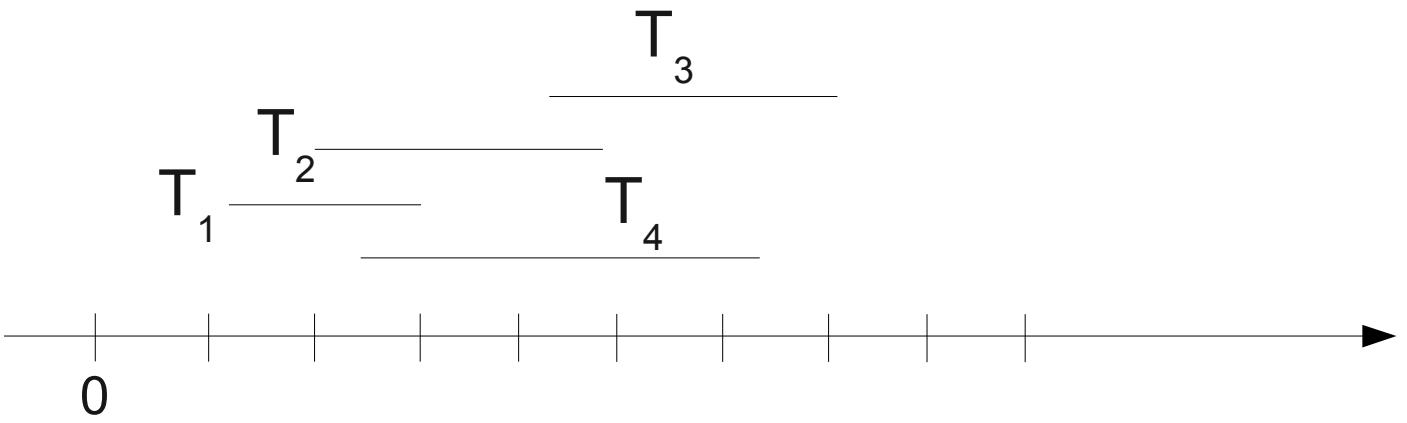
- Escalonando tarefas no tempo (*interval scheduling*) com pesos
- Programação Dinâmica

Escalonamento de Tarefas

- Conjunto de N tarefas a serem executadas
- Cada tarefa tem horário para iniciar e terminar

Exemplo

- N=4, T_1 = [1, 3], T_2 = [2, 5], T_3 = [4, 7], T_4 = [2.5, 6]



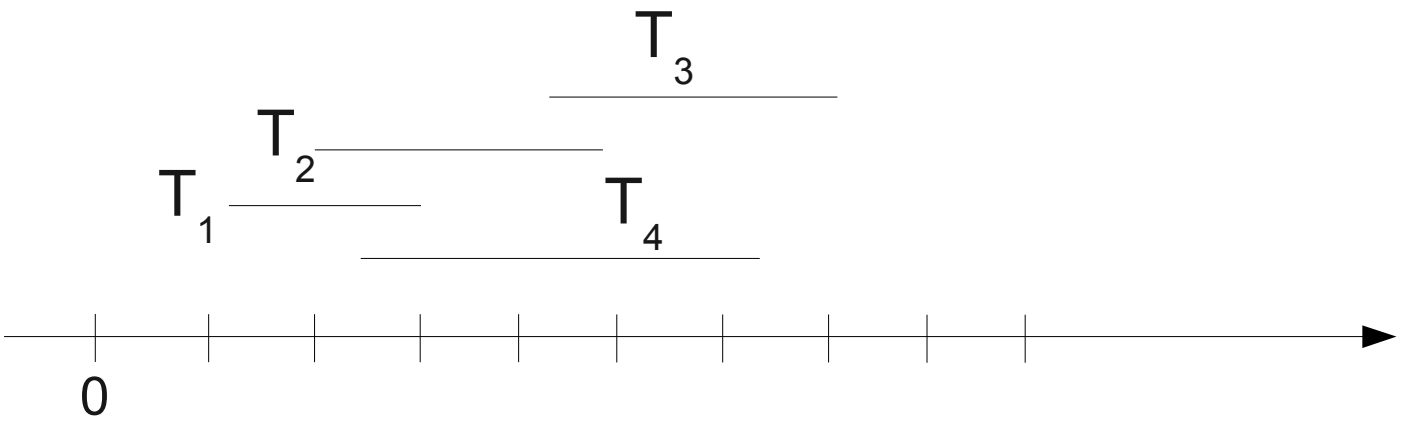
- **Problema:** Quais tarefas executar de forma a maximizar número de tarefas?

Variação com Pesos

- Tarefas possuem valores diferentes
  - pesos associados às tarefas

Exemplo

- N=4, T_1 = [1, 3], T_2 = [2, 5], T_3 = [4, 7], T_4 = [2.5, 6]
  $v_1 = 2$, $v_2 = 5$, $v_3 = 2$, $v_4 = 3$



- **Problema:** Quais tarefas executar de forma a maximizar o valor das tarefas?

Variação com Pesos

Algoritmo antigo funciona?

- Algoritmo guloso, escolhendo tarefas que terminam mais cedo primeiro que não tenham conflito

- **Novas idéias?**
- Escolher mais valiosa primeiro?

Notação

- c(i) : instante em que tarefa i inicia
- f(i) : instante em que tarefa i termina
- Assumir que tarefas estão ordenadas pelo tempo de término
  - f(1) < f(2) < ... < f(n)
  - podemos ordená-las se for necessário
- p(i) : tarefa de maior índice menor do que i que não colide com i
  - primeira tarefa antes de i que pode ser escalonada quando i é escalonada

Algoritmo Para o Problema

- Vamos estudar a *estrutura* da solução ótima
- Considere O o conjunto de tarefas ótimo
  - tarefas que maximizam o valor do escalonamento
- Ex. O = $\{T_2, T_5, ..., T_7\}$, com n = 10
- O que podemos dizer da última tarefa, T_n?
- Duas possibilidades
  - Pertence a O
  - Não pertence a O

Analisando Solução Ótima

- Se T_n pertence a O, então nenhuma tarefa que colide com T_n pode pertencer a O
  - Pois caso contrário, teríamos uma colisão
  - mais ainda, nenhuma tarefa no intervalo entre o fim de p(n) e f(n) pertence a O
- Se T_n pertence a O, então O é solução ótima para o problema com tarefas 1 até p(n)
  - Caso contrário, escolher outras tarefas e melhorar a solução

Importantes observações!

Analisando Solução Ótima

- Se T_n não pertence a O, então O é a solução ótima para o problema com tarefas T_1 até T_{n-1}
  - Pois caso contrário, escolher outras tarefas T_1 até T_{n-1} e melhorar a solução sem T_n

Importantes observações!

- Como podemos usar estas observações?

Decompondo a Solução

- **Idéia:** construir solução ótima a partir de soluções ótimas para problemas menores
- Considere as tarefas T_1, T_2 , … , T_j
- O_j é a solução ótima quando consideramos j tarefas
- OPT(j) é o *valor* da solução O_j
  - OPT(j) = soma dos valores das tarefas em O_j
- Dado as observações, qual valor da solução ótima quando consideramos j tarefas?

Custo da Solução Ótima

- Duas possibilidades:
  - j pertence ou não a solução ótima O_j
- Supor que j pertence a solução ótima O_j
  - $OPT(j) = OPT(p(j)) + v_j$
- Supor que j não pertence a O_j
  - $OPT(j) = OPT(j-1)$
- Qual das duas será utilizada?
  - a de maior valor, pois queremos maximizar
- Ou seja
  - $OPT(j) = \max (OPT(p(j)) + v_j , OPT(j-1))$

Recursão do Valor Ótimo

- OPT(j) = max (OPT(p(j)) + v_j , OPT(j-1))
- Algoritmo para calcular OPT(n) ?
  - Valor ótimo do problema inicial

Algoritmo Recursivo

```
Compute-OPT(j)
Se j = 0 entao
   retorna 0
Caso constrario
   retorna max (Compute-OPT(p(j)) + v(j),
                Compute-OPT(j-1))
```

Complexidade

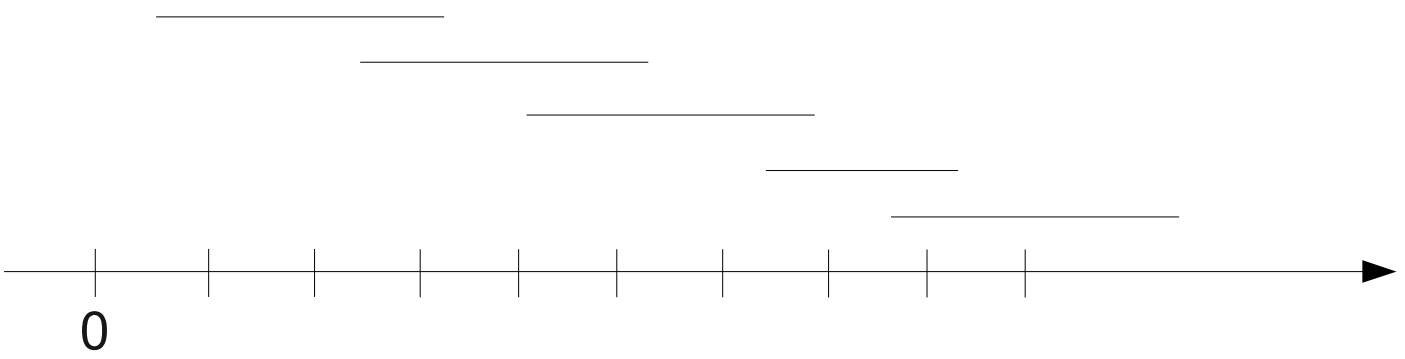
- Algoritmo recursivo

```
Compute-OPT(j)
Se j = 0 entao
   retorna 0
Caso contrario
   retorna max (Compute-OPT(p(j)) + v(j),
                Compute-OPT(j-1))
```

- Custo é o número de chamadas à função
- Árvore de execução pode ser muito grande
  - Exponencial no número de tarefas
- Exemplo?

Custo Exponencial

- Exemplo



- Árvore de execução?

Melhorando o Algoritmo

- Observação da árvore de execução
- **Problema**: mesmo cálculo feito inúmeras vezes sem necessidade
- Quantos cálculos de OPT são necessários?
- n, pois OPT(n) pode ser construído recursivamente
- Como melhorar algoritmo?

Armazenar valores já computados!

Outro Algoritmo

- Usar um vetor para armazenar valores
- M[j] : custo da solução ótima considerando as j primeiras tarefas

```
M-Compute-OPT(j)
Se j = 0 entao
   retorna 0
Caso contrario
   Se M[j] nao estiver vazio entao
       retorna M[j]
   Caso contrario
      M[j] = max (M-Compute-OPT(p(j)) + vj,
                  M-Compute-OPT(j-1))
      retorna M[j]
```

- Complexidade?

Algoritmo não Recursivo

- Vetor M contém a solução para os subproblemas
  - M[n] a solução final
- **Idéia**: construir M iterativamente

```
Iterativo-Compute-OPT(j)
M[0] = 0
Para j = 1, 2, ..., n
   M[j] = max (M[p(j)] + vj, M[j-1])
```

- Complexidade?

Programação Dinâmica

- Técnica para construção de algoritmos
- Baseada na análise e decomposição da estrutura do problema
- Recursão da solução ótima
- Memorização de resultados temporários

Técnica avançada e poderosa!