M415 : Programmation dynamique

S. PERENNES., M. SYSKA

DUT INFO - IUT Nice Côte d'Azur

15 avril 2022

Cette présentation est adaptée de J. Kleinberg and E. Tardos. *Algorithm Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

Introduction

Soit un problème d'optimisation et sa solution *force brute* au coût exponentiel :

- on ne trouve pas toujours (en fait rarement) un algorithme *glouton* qui calcule la solution optimale du problème
- une autre stratégie comme *diviser pour régner* permet d'accélerer les traitements mais pas de basculer d'un temps de traitement exponentiel à un temps polynomial.

Que faire ?

Programmation dynamique

- Comme dans les stratégies *diviser pour régner* déjà vues :
  - décomposer l'espace des solutions à explorer en sous-problèmes
  - construire la solution globale par composition des solutions exactes des sous-problèmes

  Mais avec une approche inverse : du bas vers le haut (petits vers plus gros)
- Comme en *force brute* :
  - donner la solution optimale
  - explorer l'espace de toutes les solutions (à l'opposé du *glouton*)

  Implicitement et non explicitement (sinon on revient à la force brute et au coût exponentiel)

Exemple 1 : Problème du Sac à Dos

Question : Comment remplir au mieux son sac ?
Version simplifiée :

- Volume du sac V.
- Liste d'objets $O_i, i \in I$, ayant chacun un volume v_i et un prix p_i.
- **Solution Valide :** Sous ensemble d'objets $J \subset I$ vérifiant $\sum_{j \in J} v_j \leq V$.
- **Valeur de la solution :** $\sum_{j \in J} p_j$.

Autrement dit on souhaite déterminer :

$$Max\{\sum_{j \in J} p_j \text{ sous la contrainte } \sum_{j \in J} v_j \leq V\}$$

Autre application : Quelle données stocker si la capacité de stokage est limitée.

Problème du Sac à Dos, Algorithme exaustif

- Pour $k = 0, \ldots |I| - 1$.

  On génère l'ensemble S_k des sous solutions utilisant uniquement les k permier éléments.
- à l'étape k on gère au pire 2^k sous-solutions.
- $S_0 = \emptyset, S_{k+1} = \cup[s + [v_{k+1}], s], s \in S_k$ (sous reserve du respect de la condition de volume)

Remarque : Deux sous-solutions de S_k utilisant le même volume et ayant la même valeur sont Équivalentes.

Exemple : deux sous-solutions de volume 8 prix 6.

- $A = \{(5,3),(1,1),(2,2)\}$
- $B = \{(4,1),(2,1),(2,4)\}$

Problème du Sac à Dos, Prog dynamique.

Notons $V(A), P(A)$ le volume et le prix d'une solution partielle.
On ne conserve qu'une seule solution parmis les solutions équivalentes.

Idée de table

- On remplace S_k par une Table T_k qui contient les couples $(V(A), P(A))$ possibles (i.e tels que $A \in S_k$.
- La table contient une entrée unique par groupe de solutions équivalentes.
- L'algorithme doit préciser comment mettre à jour la table, c'est à dire calculer T_{k+1}.

Problème du Sac à Dos, Gestion de la table.

La table T_k contient la liste des couples (volume,prix) (V, P) valides pour toute les sous solutions utilisant les objets $0, \dots k-1$.

- $T_0 = [(0,0)]$
- **Mise à jour** $T_{k+1} = []$
  Pour $A = (V, P) \in T_k$
  - $T_{k+1} = T_{k+1} + [A]$ (on ajoute pas O_k).
  - si $V + v(O_k) \leq V$ alors
    $T_{k+1} = T_{k+1} + [V + v(O_k), P + p(O_k)]$ (on tente d'ajouter O_k).

Complexité , taille de la table.

Si les prix et les volumes sont entiers et si l'optimal vaut Opt la taille de la table est au plus

$$Opt \times V$$

Par ailleurs comme le nombre d'objets est I on effectue $|I|$ mises à jour. Donc au final la complexité est

$$\Theta(Opt V |I|)$$

En pratique on peut arrondir les prix, arrondir les volumes peut être plus délicat.

Lien avec la mémo(r)isation.

- Implicitement, pour le sac à dos on ne cherche qu'une fois si une entrée $T_k(V, P)$ existe. Chaque entrée de la table n'est générée qu'une seule fois.
- Dans certains calculs récursifs utiliser la mémorisation peut faire passer la complexité d'exponentielle à polynomiale. **Example les coefficients binomiaux** $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
  - 1
  - 11
  - 121, 1331, 14641, 1 5 10 10 5 1

  $(1 + x)^n = \sum_{k \in [0,n]} \binom{n}{k} x^k$.

Si on calcule naivement par reccurence $\binom{n}{k}$ le temps est exponentiel $\Omega(2^n)$, mais si on mémorise les calculs le temps est $\Theta(nk)$.
Autre example : les classiques *Tours de Hanoi*.

Exemple 2 : ordonnancement d'intervalles pondérés

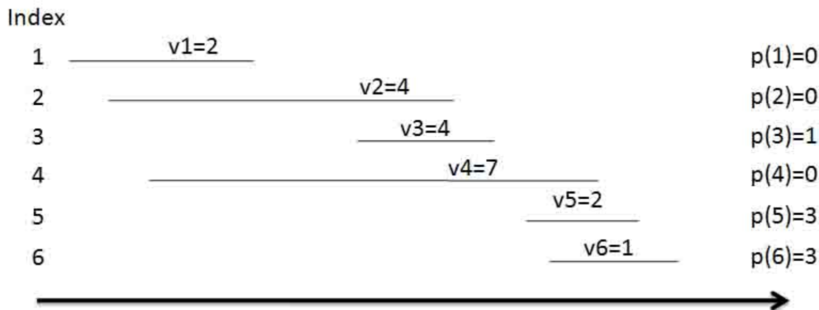
On considère :

- un ensemble de requêtes pondérées $R = \{1, 2, \cdots, n\}$
- la requête i correspond à un intervale de temps (d_i, f_i) ou s_i et la date de début et f_i la date de fin ; le poids ou valeur de la requêtes est v_i.
- (i) un sous-ensemble de requêtes est valide si les requêtes ne s'intersectent pas (deux à deux).
- le problème est de trouver un sous-ensemble de requêtes S valide (sans conflit) qui maximise la somme des valeurs : $\sum_{i \in S} v_i$

Application : Placer des taches sur une machine, la condition (i) implique qu'à chaque temps une seule requête est active.

Ordonnancement d'intervalles, approche par la date de fin.

Dans la suite on suppose que $f_1 \leq f_2 \leq \cdots \leq f_n$ et on note $p(j)$ le plus grand i, $i < j$ tel que i et j sont disjoints ($p(j) = 0$ si on n'en trouve pas).

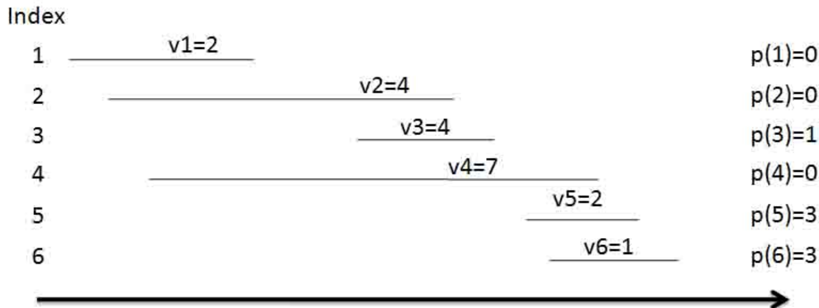


Ordonnancement d'intervalles, propriété clef.

Propriété :

Les requêtes qui ne sont pas en conflit avec la requête n sont exactement les requêtes $1, 2, \ldots, p(n)$.

Ce sont en effet les requêtes qui se terminent avant la date de début de n (d_n). Cela marche car on a ordonné les requêtes par date de fin et que l'on considère la dernière requête (n).



Exemple 2 : ordonnancement d'intervalles pondérés

Soit $\mathcal{O}$ une solution optimale du problème et considérons la dernière requête :

- soit la requête n (la dernière) appartient à $\mathcal{O}$ et dans ce cas l'optimal est la requête n union une solution optimale pour le requêtes $1, 2, \ldots, p(n)$.
- soit la requête n n'appartient pas à $\mathcal{O}$ et la solution n'est autre que l'optimal pour les requêtes $1, 2, \ldots n-1$.

Exemple 2 : ordonnancement d'intervalles pondérés

Afin de trouver la solution optimale de $\{1, 2, \cdots, n\}$ il suffit de trouver la solution de plus petits problèmes $\{1, 2, \cdots, j\}$

- $\forall j = 1 \cdots n$ on note $\mathcal{O}_j$ une solution optimale pour les requêtes $\{1, 2, \cdots, j\}$
- nous notons aussi $OPT(j)$ la valeur d'une telle solution, avec $OPT(0) = 0$
- et on cherche $\mathcal{O}_n$ de valeur $OPT(n)$

Exemple 2 : ordonnancement d'intervalles pondérés

Si on applique le raisonnement de découpage en sous-problèmes selon que la dernière requête appartient ou pas à la solution :

- si $j \in \mathcal{O}_j$ alors $OPT(j) = v_j + OPT(p(j))$
- si $j \notin \mathcal{O}_j$ alors $OPT(j) = OPT(j-1)$

Conclusion :

$$OPT(j) = max(v_j + OPT(p(j)), OPT(j-1))$$

- La formule ci-dessus permet de calculer la Table *OPT* pas à pas.
- La taille de la table est polynomiale (même linéaire).
- La notion d'équivalence implicite est que le requêtes se terminent toutes avant une certaine date.

Exemple 2 : ordonnancement d'intervalles pondérés

Il reste à décider si j appartient ou pas à $\mathcal{O}_j$.
j est dans la solution si on gagne quelque chose en la choisissant.

$j \in \mathcal{O}_j$ si et seulement si :

$$v_j + OPT(p(j)) \geq OPT(j-1)$$

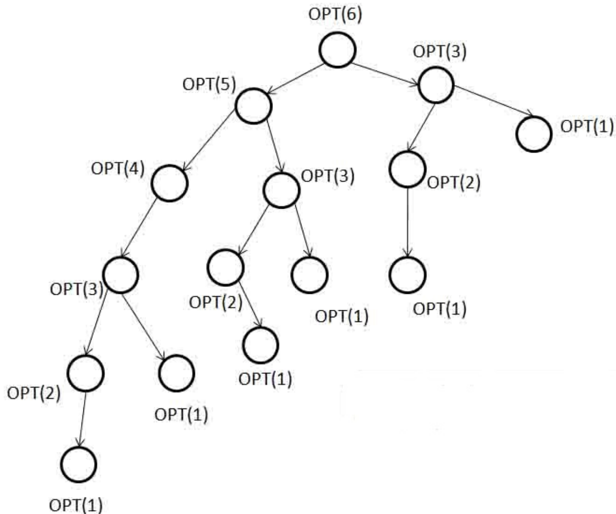
Exemple 2 : ordonnancement d'intervalles pondérés

Programmation dynamique : Donner une relation de récurrence entre les solutions des sous-problèmes qui permette de construire la solution globale et d'en déduire un algorithme récursif.
Pour un bon fonctionement il faut que le nombre de sous problèmes soit polynomial.

- trier les requêtes par date de fin
- calculer les valeurs de $p(j)$ pour tous les j
- appliquer la récurrence (return 0 si j==0)
- le nombre de sous problème est ici linéaire $1, 2, \ldots j$ pour $j \leq n$.

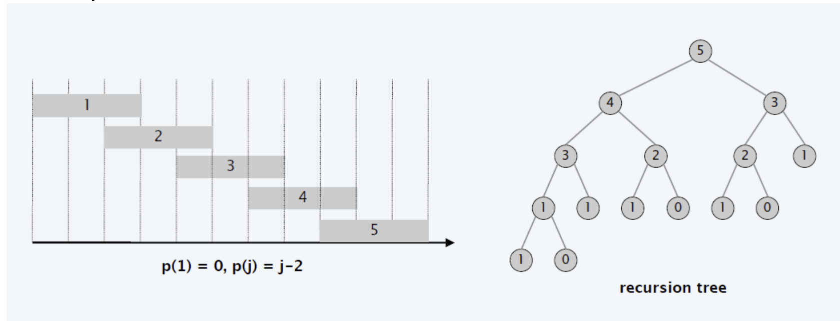
Implémentation naïve

Sur l'exemple on va obtenir l'arbre de récursion :



Implémentation naïve

Cas le pire :



L'arbre croît comme la suite de Fibonacci !

nb : erreur dans le dernier sous arbre gauche : (2,(1,0)) et non (1,(1,0))

Memoization

La solution est de mémoïser (garder en cache) les solutions des sous-problèmes. On obtient une solution en $O(n)$.

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

for j = 1 to  n
   M[j] ← empty.
M[0] ← 0.

M-Compute-Opt(j)
_________________
if M[j] is empty
   M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j – 1)).
return M[j].
```

Donner l'ensemble des intervalles de la solution

Maintenir à jour un ensemble coûte $O(n)$, il vaut mieux récupérer les intervalles une fois que le calcul est terminé.

```
Find-Solution(j)
----------------
if j = 0
   return ∅.
else if (v[j] + M[p[j]] > M[j-1])
   return { j } ∪ Find-Solution(p[j]).
else
   return Find-Solution(j-1).
```

Dérécursiviser l'algorithme

BOTTOM-UP (n, s_1, ..., s_n, f_1, ..., f_n, v_1, ..., v_n)

Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

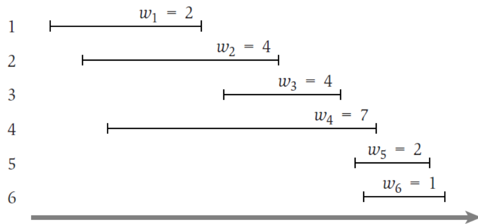
Compute $p(1), p(2), \ldots, p(n)$.

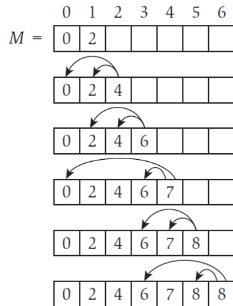
$M[0] \leftarrow 0$.

FOR j = 1 TO n

$\quad M[j] \leftarrow \max \{ v_j + M[p(j)], M[j-1] \}$.

Exemple





Exemple 3 : Les plus courts chemins

Contexte : Graphe pondéré $G = (V, E)$ où chaque arête $e \in E$ (on note $e = [u, v]$) est munie d'une longueur positive $l(e)$.

- Étant donné un sommet source v_0 on cherche le plus court chemin allant de v_0 à u pour tout $u \in V$.
- **Table & étiquettes :** Pour chaque sommet u on maintient $T_k(u)$= longueur du plus court chemins utilisant au plus k arêtes reliant v_0 à u.

Propriété

$$T_{k+1}(u) = Min\{T_k(v) + l([v, u]), v \text{ est voisin de u} \cup T_k(u)\}$$

Initialisation :

- $T_0(v) = +\infty$ pour $u \neq v_0$.
- $T_0(v_0) = 0$.

On calcule la table T_{k+1} en utilisant la propriété.

Exemple 3 : Les plus courts chemins

- La table T_k est de taille $|V|$.
- L'algorithme se termine pour $k \leq |V|$.
- Optimisation : On peut terminer dès qu'aucune valeur de T_k ne change (souvent bien avant $|V|$).
- Implicitement les chemins venant de v_0 et arrivant en v sont considérés comme équivalents et on conserve dans la table la valeur associée au meilleur.
- Si on veut récupérer les plus courts chemins il faut les stocker dans la table.

Problème bi-critère

Si on utilise deux critères : Prix, Temps (pex) la table est bien plus grande car on stocke pour tout temps d'arrivée t $T_k(u,t)$ le prix minimum pour arriver en u avant t en utilisant au plus k arêtes.