Programmation dynamique

M1 MIAGE Nanterre, Claire Hanen

2 mai 2008

1 Introduction

La programmation dynamique est une méthode de résolution exacte qui peut-être appliquée à certaines classes de problèmes d'optimisation déterministe ou stochastique. On en verra ici seulement une partie, la plus simple, mais qui a beaucoup d'applications.

Considérons tout d'abord l'application de cette méthode au problème du sac à dos.

On considère donc un ensemble $O = \{1, \ldots, n\}$ de n objets, chaque objet i possède un poids p_i et une utilité w_i. On se donne un entier P, et l'on recherche à constituer un sac avec une partie des objets de sorte que la somme des poids des objets choisis soit inférieure ou égale à P, et que la somme des utilités des objets choisis soit maximale.

On va plonger ce problème dans une famille de problèmes d'optimisation de nature similaire.

1.1 Sous-problèmes

Définissons $F_k(E) =$ l'utilité maximale d'un sac de poids inférieur ou égal à E sur l'ensemble d'objets $O_k = \{k, \ldots, n\}$.

Autrement dit, $F_k(E)$ est la solution optimale du problème d'optimisation suivant :

$$\Pi_k(E) : \begin{cases} \max \sum_{i=k}^{n} w_i x_i & \\ \sum_{i=k}^{n} p_i x_i & \leq E \\ \forall i \in \{k, \ldots, n\} & x_i \in \{0,1\} \end{cases}$$

On peut voir que notre problème initial consiste à calculer $F_1(P)$.

La méthode de résolution consiste à observer que les valeurs $F_k(E)$ sont soit très faciles à caluler, soit se déduisent les unes des autres.

1.2 Calcul de $F_n(E)$

Par définition $F_n(E)$ est l'utilité maximale d'un sac de poids inférieur ou égal à E sur l'ensemble d'objets $O_n = \{n\}$.

Autrement dit, on a un seul objet, l'objet n du problème de départ, et l'on se demande quel est la meilleure chose à faire : prendre cet objet ou non.

Il est clair que si on peut prendre l'objet, l'utilité du sac sera plus grande. D'où :

$$F_n(E) = \begin{cases} w_n & si \quad p_n \leq E \\ 0 & sinon \end{cases}$$

1.3 Equation de récurrence

Supposons maintenant que $k < n$.On peut prouver de plus l'équation de récurrence suivante :

$$F_k(E) = \begin{cases} F_{k+1}(E) & si \quad p_k > E \\ \max(F_{k+1}(E), F_{k+1}(E - p_k)) & sinon \end{cases}$$

En effet, notons $x_k^*, \ldots x_n^*$ la solution optimale du problème $\Pi_k(E)$. On a donc par définition $F_k(E) = \sum_{i=k} w_i x_i^*$.

On a deux cas possibles :

Cas n°1 : $x_k^* = 0$

Autrement dit, dans la solution optimale du problème $\Pi_k(E)$, l'objet k n'est pas dans le sac. On en déduit que $x_{k+1}^*, \ldots x_n^*$ définit un sac parmi les objets $O_{k+1} = \{k+1, \ldots, n\}$ de poids inférieur ou égal à E. C'est donc une solution réalisable du problème $\Pi_{k+1}(E)$. Or, une solution réalisable a une utilité totale inférieure ou égale à l'utilité maximale du problème $\Pi_{k+1}(E)$ qui par définition est $F_{k+1}(E)$. D'où

$$F_k(E) = \sum_{i=k+1} w_i x_i^* \leq F_{k+1}(E)$$

Cas n°2 : $x_k^* = 1$

Ce cas ne peut se produire que si $p_k \leq E$. Dans la solution optimale du problème $\Pi_k(E)$, l'objet k est dans le sac. On sait que $\sum_{i=k} p_i x_i^* \leq E$. D'où $\sum_{i=k+1} p_i x_i^* \leq E - p_k$. On en déduit que dans ce cas, $x_{k+1}^*, \ldots x_n^*$ est une solution réalisable du problème $\Pi_{k+1}(E - p_k)$.

D'où $\sum_{i=k+1} w_i x_i^* \leq F_{k+1}(E - p_k)$. On en déduit que

$$F_k(E) \leq w_k + F_{k+1}(E - p_k)$$

De ces deux cas, on déduit que si $p_k \leq E$

$$F_k(E) \leq \max(F_{k+1}(E), w_k + F_{k+1}(E - p_k))$$

Montrons maintenant la réciproque.

Soit $\hat{x}_{k+1}, \ldots, \hat{x}_n$ la solution optimale du problème $\Pi_{k+1}(E)$. Donc par définition

$$\sum_{i=k+1} w_i \hat{x}_i = F_{k+1}(E).$$

Définissons $\hat{x}_k = 0$. Il est clair que $\hat{x}_k, \ldots, \hat{x}_n$ définit un sac de poids inférieur ou égal à E, solution réalisable du problème $\Pi_k(E)$. D'où

$$F_{k+1}(E) = \sum_{i=k} w_i \hat{x}_i \leq F_k(E)$$

.

Considérons maintenant Soit $\overline{x_{k+1}}, \ldots, \overline{x_n}$ la solution optimale du problème $\Pi_{k+1}(E - p_k)$. Définissons $\overline{x_k} = 1$.

On a $\sum_{i=k}^{n} p_i \overline{x_i} = p_k + \sum_{i=k+1}^{n} p_i \overline{x_i} \leq p_k + E - pk = E$.

On en déduit que $\overline{x_k}, \ldots, \overline{x_n}$ est une solution réalisable de $\Pi_k(E)$. Par conséquent :

$$\sum_{i=k} w_i \overline{x_i} = w_k + \sum_{i=k+1} w_i \overline{x_i} = w_k + F_{k+1}(E - p_k) \leq F_k(E)$$

1.4 Algorithme de calcul

Pour calculer $F_1(P)$, nous all 2053, nous all 2053, n.1.5T3-1.5TD[

1.5 Application numérique

$P = 8$:

i	1	2	3	4	5
p_i	3	4	5	4	2
w_i	15	18	20	12	6

$E \mid i$	1	2	3	4	5
0	0	0	0	0	0
1	0	0	0	0	0
2	6	6	6	6	6
3	15	6	6	6	6
4	15	18	12	12	6
5	21	20	20	12	6
6	24	24	20	18	6
7	33	26	26	18	6
8	35	30	26	18	6

L'algoritme retournera donc la valeur 35.

1.6 Calcul de la solution optimale

L'algorithme que nous avons vu permet de calculer l'utilité maximale d'un sac. Mais il est également indispensable de connaître une solution (c'est à dire un sac) qui permet d'atteindre cette utilité maximale.

Le calcul des $F_k(E)$, combiné à l'analyse de l'équation de récurrence, va nous permettre de répondre à cette question.

En effet, on a vu que pour tout $k < n$, soit $F_k(E) = F_{k+1}(E)$, et dans ce cas, le sac optimal solution du problème $\Pi_k(E)$ est la solution optimale du problème $\Pi_{k+1}(E)$, donc l'objet k n'en fait pas partie, soit $F_k(E) = w_k + F_{k+1}(E - p_k)$, et la solution optimale du problème $\Pi_k(E)$ consiste à prendre l'objet k et la solution optimale du problème $\Pi_{k+1}(E - p_k)$.

Cette propriété permet de reconstituer itérativement la solution optimale à partir du tableau des F_k et de la valeur obtenue $F_1(P)$. L'algorithme ci-dessous construit $x[1], \ldots, x[n]$ sac optimal. Pour notre exemple numérique, voici le déroulement de cet algorithme, en donnant la valeur de E au début de l'itération k :

k	1	2	3	4	5
E	8	5	5	0	0
$x[k]$	1	0	1	0	0

2 Quels problèmes ?

Pour pouvoir résoudre un problème par la programmation dynamique, il faut pouvoir le représenter et le formuler selon un vocabulaire précis. Considérons

Algorithm 2 Construction du sac optimal

$E = P$
pour $k = 1$ *à* $n - 1$ **faire**
  si $F[k][E] = F[k+1][E]$ **alors**
    $x[k] = 0$
  sinon
    $x[k] = 1 \quad E = E - p[k]$
si $F[n][E] = 0$ **alors**
  $x[n] = 0$
sinon
  $x[n] = 1$

Retourner x

une formulation habituelle, où $Opt = \max$ ou $Opt = \min$:

$$\begin{array}{l} \text{½} \\ \quad Opt f(X) \\ \quad X \in D \end{array}$$

2.1 Etats, phases, décisions

2.1.1 Décisions, politique

La première chose est qu'une solution X quelconque doit pouvoir être interprétée comme une suite de décisions.

Par exemple, on peut voir un sac à dos comme n décisions successives : prendre ou ne pas prendre l'objet 1, ..., prendre ou ne pas prendre l'objet n.

On appelle *horizon* H le nombre de décisions conduisant à une solution. Pour le sac à dos, on a donc $H = n$, le nombre d'objets. On appelle traditionnellement *politique* une suite de H décisions.

2.1.2 Phases

Une phase correspond à une étape du processus de décision. La phase k se situe après les $k - 1$ premières décisions.

2.1.3 Etats

La construction d'une solution par une suite de décisions doit pouvoir être décrite comme agissant sur l'état du système.

Ainsi, l'état du système au début de la phase k, après les $k - 1$ premières décisions, doit pouvoir être décrit au moyen d'une quantité qui résume l'effet des décisions du passé sur les décisions possibles ultérieures.

Cette quantité peut être mono ou pluri-dimensionnelle. Bien entendu, elle peut être simplement le vecteur des $k - 1$ premières décisions. Mais dans cer-

tains cas (ceux où la programmation dynamique a sa place comme méthode de résolution), elle peut être beaucoup plus simple.

Dans l'exemple du sac à dos, pour savoir quelles décisions on peut prendre à partir de la phase k pour construire un sac satisfaisant la contrainte de poids, on a besoin de savoir uniquement quel est le poids encore disponible dans le sac, et pas quels objets on a mis dedans dans les phases précédentes.

Ainsi, on doit avoir la propriété suivante :

Si $\delta_k, \ldots, \delta_H$ est une suite de décisions faisables à partir de l'état E du système à la fin de la phase $k-1$, alors si l'on prend n'importe quelle suite de décisions $\delta_1, \ldots, \delta_{k-1}$ qui met le système dans l'état E, $\delta_1, \ldots, \delta_{k-1}, \delta_k, \ldots, \delta_H$ est une politique faisable (correspondant à une solution du problème).

On doit également pouvoir décrire le ou les états initiaux du système au début de la phase 1.

Dans la suite, on note $\mathcal{E}_k$ l'ensemble des états possibles du système à la fin de la phase k, et $\mathcal{E}_0$ l'ensemble des états possibles au début de la phase 1. Le plus souvent, cet ensemble est réduit à un singleton.

Dans le problème du sac à dos, on a : $\mathcal{E}_0 = \{P\}$ et

$$\forall k \in \{1, \ldots, n\}, \quad \mathcal{E}_k = \{0, \ldots, P\}.$$

2.1.4 Transitions

L'effet d'une décision sur l'état du système (une fois définie la notion d'état adaptée au problème que l'on traite) doit pouvoir être décrit de manière fine.

On décrit pour cela tout d'abord, pour chaque couple $(k, E), k \in \{1, \ldots, H\}, E \in \mathcal{E}_{k-1}$, l'ensemble des décisions possibles à la phase k sachant que le système est dans l'état E. Cet ensemble est noté $\mathcal{D}_k(E)$.

Dans le cas du sac à dos, on a vu qu'à la phase k, si l'objet k est trop gros, une seule décision est possible : ne pas le prendre, alors que s'il est suffisament petit ($p_k \leq E$), on peut soit le prendre, soit ne pas le prendre.

Notons δ_k^0 la décision de ne pas prendre l'objet k, et δ_k^1 la décision de prendre l'objet k. On a donc :

$$\mathcal{D}_k(E) = \begin{array}{ll} \{\delta_k^0\} & si \quad p_k > E \\ \{\delta_k^0, \delta_k^1\} & sinon \end{array}$$

NB : dans les exemples traités dans ce cours, les décisions sont toujours en nombre fini. Mais rien n'interdit d'avoir des ensembles $\mathcal{D}_k(E)$ infinis.

Une fois décrit l'ensemble des décisions possibles, on va décrire les transitions, c'est à dire comment une décision va modifier l'état du système.

Ainsi, si au début de la phase k, le système est dans l'état $E \in \mathcal{E}_{k-1}$, et si une décision particulière $\delta \in \mathcal{D}_k$ est choisie, à la fin de la phase k, le système sera donc dans un état $E' = \delta | E \in \mathcal{E}_k$.

Pour le sac à dos,on a $\delta_k^0 | E =$Notons

2.1.5 Fonctions de coût et fonction objectif

La résolution d'un problème à l'aide de la programmation dynamique nécessite, au delà des notions vues précédemment que la fonction objectif ait de bonnes propriétés.

Il existe des descriptions très générales de ces propriétés qui permettent d'appliquer la méthode à des problèmes très variés. toutefois, ici, nous allons par souci de simplification nous restreindre à une sous-classe de ces fonctions.

On supposera qu'à chaque décision possible δ prise dans l'état E à une phase k est associé un coût (ou un profit), appelé coût immédiat (ou profit immédiat). On le notera $c(\delta, E, k)$ - on pourra éventuellement enlever des paramètres si le coût ne dépend pas de la phase ou de l'état.

Si $\delta_1, \ldots, \delta_H$ est une solution réalisable du problème,conduisant aux états intermédiaires $E_0, E_1, \ldots, E_H$ les fonctions considérées seront de la forme suivante, sachant qu'en cas de minimisation on considérera des coûts et en cas de maximisation des profits :

$$f(\delta_1, \ldots, \delta_H) = \sum_{k=1}^{H} c(\delta_k, E_{k-1}, k)$$
$$f(\delta_1, \ldots, \delta_H) = \Pi_{k=1}^{H} c(\delta_k, E_{k-1}, k)$$
$$f(\delta_1, \ldots, \delta_H) = \min_{k=1,\ldots,H} c(\delta_k, E_{k-1}, k)$$
$$f(\delta_1, \ldots, \delta_H) = \max_{k=1,\ldots,H} c(\delta_k, E_{k-1}, k)$$

Autrement dit, somme ou produit des coûts/profits immédiats,min ou max de ces coûts ou profits.

Dans la suite, on notera $S(a_1, \ldots a_i)$ la somme des paramètres, $P(a_1, , \ldots a_i)$ le produit, $m(a_1, \ldots a_i)$ le min et $M(a_1, \ldots a_i)$ le max.

Dans le cas du sac à dos, le profit immédiat de la décision δ_k^0 ne dépend pas de l'état de départ, et vaut 0. Le profit immédiat de la décision δ_k^1 vaut w_k (utilité de l'objet k).

La fonction objectif à maximiser est bien la somme des profits immédiats d'une suite de décisions.

2.2 Schéma de programmation dynamique arrière

Si un problème peut être représenté à l'aide des éléments ci-dessus, alors on peut définir un algorithme de programmation dynamique pour le résoudre. Cet algorithme va reprendre les éléments que nous avons déjà étudiés pour le problème du sac à dos, mais dans un contexte beaucoup plus général.

2.2.1 F_k

On appelle sous-politique à partir de l'état E et de la phase k une suite de décisions admissibles $\delta_k, \ldots, \delta_H$.

Ainsi, pour le problème du sac à dos, une sous-politique à partir de l'état E et de la phase k est en fait un sac à dos parmi les objets $\{k, \ldots, n\}$ de poids total inférieur à E, puisque E représente le poids restant disponible dans le sac partiellement rempli.

Si π est une sous-politique , on note $C(\pi)$ la somme, le produit, le max ou le min des profits ou coûts immédiats des décisions de cette sous-politique, selon la fonction objectif de départ.

Soit $E \in \mathcal{E}_{k-1}$. On définit $F_k(E)$ comme le profit maximal (ou le coût minimal) d'une sous-politique à partir de l'état E et de la phase k.

Pour le problème du sac à dos, on retrouve bien la définition donnée au début : $F_k(E)$ est l'utilité maximale (somme des utilités) d'un sac parmi les objets $\{k, \ldots, n\}$ de poids total inférieur à E.

2.2.2 Initialisation

Afin de pouvoir définir une récurrence, comme dans le cas du sac à dos, on doit tout d'abord s'assurer que le calcul de $F_H(E)$, pour toutes les valeurs de $E \in \mathcal{E}_{H-1}$ est simple. Notons que ce problème n'a plus qu'une variable, l'unique décision à prendre à la dernière phase. Ainsi, dans le cas d'une minimisation, ce problème se formule ainsi :

$$F_H(E) = \min_{\delta \in \mathcal{D}_H(E)} c(\delta, E, H)$$

Dans le cas d'une maximisation, il suffit de remplacer min par max.

Nous avons vu que le problème est particulièrement simple à résoudre dans le cas du sac à dos puisque l'ensemble contient une ou deux décisions.

2.2.3 Equation de récurrence arrière

On donne ici la version la plus générale, où g est l'une des fonctions S, P, m, M, dans le cas d'une minimisation (pour une maximisation, remplacer min par max) :

Theorème 2.1.

$$\forall E \in \mathcal{E}_{k-1}, F_k(E) = \min_{\delta \in \mathcal{D}_k(E)} g(c(\delta, E, k), F_{k+1}(\delta|E)) \tag{1}$$

Dans le cas du sac à dos, g est la fonction somme,$\mathcal{D}_k(E) = \{\delta_k^0\}$ si $p_k > E$, d'où dans ce cas une seule décision possible de profit immédiat 0, avec $\delta_k^0|E = E$ et donc $F_k(E) = F_{k+1}(E)$. Lorsque $p_k \leq E$, on a $\mathcal{D}_k(E) = \{\delta_k^0, \delta_k^1\}$ de profits respectifs 0 et w_k et d'états résultants E et $E - p_k$ respectivement. D'où $F_k(E) = \max(F_{k+1}(E), w_k + F_{k+1}(E - p_k))$.

2.2.4 Preuve de l'équation de récurrence

les fonctions considérées ici S, P, m, M ont la propriété suivante :

Propriété 2.2.

$$g(a_1, \ldots, a_i) = g(a_1, g(a_2, \ldots, a_i)) \forall i \geq 2$$

Soit $\pi^* = (\delta_k^*, \ldots, \delta_H^*)$ la sous-politique optimale à partir de l'état E, c'est à dire telle que $F_k(E) = g(\pi^*)$.

A cause de la propriété de g (propriété 2.2), on sait que $C(\pi^*) = g(c(\delta_k^*, E, k), C(\delta_{k+1}^*, \ldots, \delta_H^*))$.

Or $\delta_{k+1}^*, \ldots, \delta_H^*$ est une sous-politique admissible à partir de l'état $\delta_k^*|E$ et la phase k+1. Par conséquent, dans le cas d'une minimisation, on a $C(\delta_{k+1}^*, \ldots, \delta_H^*) \geq F_{k+1}(\delta_k^*|E)$, dans le cas d'une maximisation, on a l'inégalité inverse ($\leq$).

Maintenant g a également la propriété suivante :

Propriété 2.3. *Si* $c \geq b \geq 0, \quad g(a, b) \leq g(a, c)$

On le vérifie aisément pour la somme, le produit, le min et le max.

On en déduit que $C(\pi^*) \geq g(c(\delta_k^*, E, k), F_{k+1}(\delta_k^*|E))$ (inégalité $\leq$ dans le cas d'une maximisation).

Or

$$g(c(\delta_k^*, E, k), F_{k+1}(\delta_k^*|E)) \geq \min_{\delta \in \mathcal{D}_k(E)} g(c(\delta, E, k), F_{k+1}(\delta|E))$$

de même

$$g(c(\delta_k^*, E, k), F_{k+1}(\delta_k^*|E)) \leq \max_{\delta \in \mathcal{D}_k(E)} g(c(\delta, E, k), F_{k+1}(\delta|E))$$

D'où $F_k(E) \geq \min_{\delta \in \mathcal{D}_k(E)} g(c(\delta, E, k), F_{k+1}(\delta|E))$ dans le cas d'une minimisation, et

$F_k(E) \leq \max_{\delta \in \mathcal{D}_k(E)} g(c(\delta, E, k), F_{k+1}(\delta|E))$

Dans le cas d'une maximisation.

Montrons maintenant la réciproque, c'est à dire que, dans le cas d'une minimisation :

$$\forall \delta \in \mathcal{D}_k(E), \quad g(c(\delta, E, k), F_{k+1}(\delta|E)) \geq F_k(E)$$

et dans le cas d'une maximisation, l'inégalité inverse($\leq$).

Considérons pour cela une décision quelconque $\delta \in \mathcal{D}_k(E)$. Soit π la sous-politique optimale à partir de la phase $k+1$ dans l'état $\delta|E$, c'est à dire telle que $C(\pi) = F_{k+1}(\delta|E))$.

Définissons $\pi' = (\delta, \pi)$. Par définition des états et des transitions, π' est une politique réalisable à partir de l'état E et de la phase k. Par conséquent, dans le cas d'une minimisation, $C(\pi') \geq F_k(E)$ ($\leq$ dans le cas d'une maximisation). Or $C(\pi') = g(c(\delta, E, k), C(\pi)) = g(c(\delta, E, k), F_{k+1}(\delta|E))$, à cause de la propriété 2.2 de g.

D'où l'on déduit le résultat.

2.2.5 Algorithme

De l'équation de récurrence et de l'initialisation on déduit un algorithme qui peut s'exprimer ainsi de manière générique pour calculer la valeur optimale recherchée qui est max ou min de $F_1(E), E \in \mathcal{E}_0$.

NB : dans le cas du sac à dos, il n'y a qu'un état initial, l'état $E_0 = P$, donc la valeur recherchée est $F_1(P)$.

Algorithm 3 Calcul des F_k dans le cas général

pour chaque $E \in \mathcal{E}_{H-1}$ **faire**
  $F_H(E) = \min_{\delta \in \mathcal{D}_H(E)} c(\delta, E, H)$ (ou max) pour une maximisation

pour $k = H-1$ *à* 1 **faire**
  pour chaque $E \in \mathcal{E}_{k-1}$ **faire**
    $F_k(E) = \min_{\delta \in \mathcal{D}_k(E)} g(c(\delta, E, k), F_{k+1}(\delta|E))$ (ou max)

Retourner $\min\limits_{E \in \mathcal{E}_0} F_1(E)$ (ou max)

On peut également, comme pour le sac à dos, produire la politique optimale $\delta_1^*, \ldots, \delta_H^*$ à partir du calcul ci-dessus.

Algorithm 4 Calcul de la politique optimale

E =état initial correspondant à la valeur retournée par l'algorithme 2.2.5
pour $k = 1$ *à* $H-1$ **faire**
  Chercher $\delta \in \mathcal{D}_k(E)$ tel que $F_k(E) = g(c(\delta, E, k), F_{k+1}(\delta|E))$
  $\delta_k^* = \delta$
  $E = \delta|E$

Chercher $\delta \in \mathcal{D}_{H-1}(E)$ tel que $F_H(E) = c(\delta, E, H)$
$\delta_H^* = \delta$

2.2.6 Complexité

La complexité d'une méthode de programmtion dynamique est en fait liée à trois éléments importants :

- Le nombre de phases (H)
- le nombre maximal d'états par phases (N)
- le nombre maximal de décisions par phases, (D) ou, dans le cas de décisions infinies, la complexité du calcul de l'équation de récurrence.

Lorsque D est fini, la complexité sera en $O(HND)$.

La taille mémoire nécessaire pour le calcul est, si l'on stocke la matrice en entier $H \times N$. Il est possible de réduire l'espace mémoire afin de calculer la valeur de l'optimum, en ne stockant que deux colonnes de la matrice à chaque fois. Mais dans ce cas, on ne peut pas reconstituer la solution optimale à l'aide de l'algorithme 4.

Pour le problème du sac à dos, $H = n, N = P+1, D = 2$ d'où une complexité $O(nP)$.

2.3 Chemins dans le Graphe d'état

L'ensemble des éléments permettant de construire une méthode de programmation dynamique pour résoudre un problème peut être représenté sous forme d'un graphe.

Les sommets de ce graphe sont les couples (k, E) où k est un numéro de phase, et $E \in \mathcal{E}_{k-1}$, ainsi qu'un ensemble de sommets $(H+1, E)$, avec $E \in \mathcal{E}_H$

Les sommets sont reliés par des arcs correspondant aux transitions. Ainsi, pour chaque $\delta \in \mathcal{D}_k$ on a un arc entre (k, E) et $(k+1, \delta|E)$. On peut valuer cet arc par le coût/profit immédiat de la décision $c(\delta, E, k)$.

Ainsi, une politique admissible π correspond à un chemin entre un sommet $(1, E)$ et un sommet $(H+1, E')$.

Le coût/profit de cette politique $C(\pi)$ correspond alors à la valeur du chemin avec l'une des fonctions g choisie (somme, produit,min ou max des valeurs des arcs).

Résoudre le problème, c'est donc calculer le chemin dans le graphe entre un sommet $(1, E)$ et un sommet $(H+1, E')$ de valeur minimale (ou maximale).

On peut alors réinterpréter $F_k(E)$ en termes de graphes : $F_k(E)$ est la valeur minimale (ou maximale) d'un chemin issu du sommet (k, E) et ayant pour extrémité un sommet $(H+1, E')$.

Et l'équation de récurrence de la programmation dynamique est alors celle qui est très utilisée dans la recherche d'un plus court chemin (ou d'un plus long). Pour le cas de la somme, par exemple, dans un graphe $G = (S, A, v)$,si l'on note $\lambda(x)$ la valeur minimale d'un chemin issu de x, on a pour tout sommet x,

$$\lambda(x) = \min_{y|(x,y)\in A} \lambda(y)$$

2.4 Schéma de programmation dynamique avant

La représentation en termes de graphes conduit à imaginer un autre algorithme pour calculer la solution du problème d'optimisation, dans le cas où il n'y a qu'un seul état initial $\mathcal{E}_0 = \{E_0\}$.

En effet, on recherche dans ce cas le chemin dans le graphe issu du sommet $(1, E_0)$ de valeur g minimale (ou maximale).

Définissons $G_k(E) =$ la valeur minimale (ou maximale) d'un chemin dans le graphe entre le sommet $(1, E_0)$ et $(k+1, E)$.

La solution de notre problème est $\min_{E\in\mathcal{E}_H} G_H(E)$ (ou max).

On peut également fournir une équation de récurrence, basée cette fois sur les prédecesseurs pour calculer $G_k(E)$. Pour un etat $E \in \mathcal{E}_k$, et une décision $\delta \in \mathcal{D}_k$ notons $\delta^{-1}|E$ l'état E' de $\delta \in \mathcal{D}_{k-1}$ tel que $\delta|E' = E$. Autrement dit, on a dans le graphe un arc entre (k, E') et $(k+1, E)$, correspondant à la décision δ.

Theorème 2.4.

$$G_k(E) = \min_{\delta \in \mathcal{D}_k} g(c(\delta, \delta^{-1}|E,k), G_{k-1}(\delta^{-1}|E))$$

(ou max *si problème de maximisation)*

Si l'on applique cette équation au problème du sac à dos, on obtient : pour tout couple $(k+1, E)$, ce sommet a un ou deux précédesseurs dans le graphe : toujours (k, E) correspondant à la décision de ne pas prendre l'objet k, et lorsque $E + p_k \leq P$,il a pour prédecesseur $(k, E + p_k)$.

$G_0(P) = 0$ et

$$G_k(E) = \begin{cases} G_{k-1}(E) & si \quad E + p_k > P \\ \max(G_{k-1}(E), w_k + G_{k-1}(E + p_k)) & sinon \end{cases}$$

On peut en déduire un nouvel algorithme de résolution du sac à dos fondé sur cette équation de récurrence (à écrire en exercice).

3 Exemples

Dans cette section, nous allons voir quelques exemples de schémas de programmation dynamique sur des problèmes variés.

3.1 Sac à dos bidimensionnel

Ce premier exemple a pour objectif de montrer que les états peuvent appartenir à un espace à plusieurs dimensions.

On considère un ensemble de n objets O, chaque objet i possède un poids p_i, un volume v_i, une utilité w_i. On se donne deux entiers positifs P et V. On cherche à constituer un sous-ensemble d'objets (sac) de poids total $\leq P$ et de volume total $\leq V$, qui soit d'utilité maximale.

On peut modéliser ce problème ainsi :

$$\begin{cases} \max \sum_{i=1}^{n} w_i x_i & & \\ \sum_{i=1}^{n} p_i x_i & \leq & P \\ \sum_{i=1}^{n} v_i x_i & \leq & V \\ \forall i \in \{1, \ldots, n\} & x_i \in & \{0,1\} \end{cases}$$

Si l'on considère qu'une solution du problème est constituée par la même suite de décisions que dans le problème du sac à dos : prendre ou non l'objet 1, ..., prendre ou non l'objet n.

Au début d'une phase k, pour déterminer quelle suite de décisions est possible, on a besoin de connaître le poids restant disponible dans le sac, ainsi que le volume restant.

Par conséquent, un état sera un couple (E, U), où E représente le poids restant et U le volume restant. On aura $0 \leq E \leq P$ et $0 \leq U \leq V$ à chaque phase.

Définissons les décisions possibles à la phase k dans l'état (E, U) : Si l'objet k rentre dans le sac, c'est à dire si $p_k \leq E$ et $v_k \leq U$, les deux décisions possibles sont $\{\delta_k^0, \delta_k^1\}$ (ne pas prendre k, et prendre k).

$\delta_k^0|(E, U) = (E, U)$ (on reste dans le même état), et

$\delta_k^1|(E, U) = (E - p_k, U - v_k)$.

Si l'objet ne rentre pas, une seule décision est possible, δ_k^0.

Le profit immédiat de δ_k^0 est 0, celui de δ_k^1 est w_k.

Ce schéma permet de définir automatiquement les F_k :

$$F_n(E, U) = \begin{cases} w_n & si \quad p_n \leq E \text{ et } v_n \leq U \\ 0 & sinon \end{cases}$$

$$F_k(E, U) = \begin{cases} F_{k+1}(E, U) & si \quad p_k > E \text{ ou } v_k > U \\ \max(F_{k+1}(E, U), w_k + F_{k+1}(E - p_k, U - v_k)) & sinon \end{cases}$$

D'où l'algorithme 5.

3.2 Sac à dos inversé

On considère le problème suivant : n objets, de poids respectifs p_i et de regret r_i. On se donne également un entier P. On appelle regret d'un sac la somme des regrets des objets **qui ne sont pas** dans le sac. On cherche à construire un sac de poids inférieur ou égal à P qui soit de regret minimal.

En définissant,pour tout objet i, x_i qui vaut 1 si l'objet est dans le sac et 0 sinon, on obtient le modèle suivant :

$$\begin{cases} \min \sum_{i=1}^{n} r_i(1 - x_i) & \\ \sum_{i=1}^{n} p_i x_i & \leq P \\ \forall i \in \{1, \ldots, n\} & x_i \in \{0, 1\} \end{cases}$$

Puisque l'objet i produit un regret s'il n'est pas dans le sac, c'est à dire si $1 - x_i = 1$.

On cherche à définir un schéma de programmation dynamique pour ce problème. Pour cela on définit la phase k comme étant celle du choix pour l'objet k, et l'état du système comme le poids des objets **déjà dans le sac** (une variante de la variable d'état choisie jusque là)

Les ensembles d'états $\mathcal{E}_k$ sont les entiers entre 0 et P.

A la phase k dans l'état E on peut prendre soit deux décisions (mettre l'objet k dans le sac δ_k^1 ou ne pas le mettre δ_k^0) soit une seule (δ_k^0 si k ne rentre pas, c'est à dire si $E + p_k \leq P$).

On a $\delta_k^0|E = E$ et $\delta_k^1|E = E + p_k$ (le poids des objets dans le sac augmente de p_k).

Le coût immédiat de δ_k^0 est r_k, celui de δ_k^1 est 0.

$F_k(E)$ est alors le regret minimum d'un sac de poids inférieur ou égal $P - E$ parmi les objets $\{k, \ldots, n\}$.

$F_1(0)$ est donc la solution recherchée ($E_0 = 0$).

Algorithm 5 Sac à dos bidimensionnel

pour $E = 0$ *à* P **faire**
  pour $U = 0$ *à* V **faire**
    si $p[n] \leq E$ *et* $v[n] \leq U$ **alors**
      $F[n][E][U] = w_n$
    sinon
      $F[n][E][U] = 0$

pour $k = n - 1$ *à* 1 **faire**
  pour $E = 0$ *à* P **faire**
    pour $U = 0$ *à* V **faire**
      $F[k][E][U] = F[k+1][E][U]$
      si $p[k] \leq E$ *et* $v[k] \leq U$ *et* $w[k] + F[k+1][E - p[k]][U - v[k]] > F[k+1][E][U]$ **alors**
        $F[k][E][U] = w[k] + F[k+1][E - p[k]][U - v[k]]$

optimum= $F[1][P][V]$
$E = P, U = V$
pour $k = 1$ *à* $n - 1$ **faire**
  si $F[k][E][U] = F[k+1][E][U]$ **alors**
    $x[k] = 0$
  sinon
    $x[k] = 1 \quad E = E - p[k] U = U - v[k]$

si $F[n][E][U] = 0$ **alors**
  $x[n] = 0$
sinon
  $x[n] = 1$

Retourner x et optimum

$$F_n(E) = \begin{cases} r_n & si \quad E + p_n > P \\ 0 & sinon \end{cases}$$

$$F_k(E) = \begin{cases} r_k + F_{k+1}(E) & si \quad E + p_k > P \\ \min(r_k + F_{k+1}(E), F_{k+1}(E + p_k)) & sinon \end{cases}$$

On en déduit l'algorithme 6 présenté ci-dessous.
La complexité de l'algorithme est en $O(nP)$.
Appliquons l'algorithme au problème suivant :

$P = 8$

i	1	2	3	4	5
p_i	3	4	5	4	2
r_i	15	18	20	12	6

Algorithm 6 Sac à dos inversé

pour $E = 0$ *à* P **faire**
  si $E + p[n] \leq P$ **alors**
    $F[n][E] = r[n]$
  sinon
    $F[n][E] = 0$

pour $k = n - 1$ *à* 1 **faire**
  pour $E = 0$ *à* P **faire**
    $F[k][E] = r[k] + F[k+1][E]$
    si $E+p[k] \leq P$ *et* $F[k+1][E + p[k]] < r[k] + F[k+1][E]$ **alors**
      $F[k][E] = F[k+1][E - p[k]]$

optimum= $F[1][0]$
$E = 0$
pour $k = 1$ *à* $n - 1$ **faire**
  si $F[k][E] = r[k] + F[k+1][E]$ **alors**
    $x[k] = 0$
  sinon
    $x[k] = 1 \quad E = E + p[k]$

si $F[n][E] = 0$ **alors**
  $x[n] = 1$
sinon
  $x[n] = 0$

Retourner x et optimum

$E \vert i$	1	2	3	4	5
0	36	26	12	0	0
1	38	30	12	0	0
2	47	32	18	0	0
3	50	36	18	6	0
4	53	38	26	6	0
5	56	50	32	12	0
6	65	50	32	12	0
7	71	56	38	18	6
8	71	56	38	18	6

La solution optimale est de valeur 36 et correspond au sac contenant l'objet 1 et 3.

3.3 Gestion de stock

On considère ici un problème très classique de plannification avec gestion de stock. Une entreprise produit un certain type de produit (par exemple des vélos...) en unités entières. Elle souhaite plannifier sa production sur H semaines. On suppose que toutes les livraisons aux clients se font en fin de semaine.

L'entreprise dispose sur son carnet de commande d'une demande D_i d'unités de produit pour chaque semaine i.

La capacité de production pour chaque semaine est variable d'une semaine à l'autre, et est estimée à l'avance : la semaine i, Q_i unités de produits peuvent être produites. Le coût de la production est également variable. Une unité de produit fabriquée la semaine i coûte c_i.

Il peut arriver que la demande pour la semaine i excède la capacité de production de la semaine. Dans ce cas l'entreprise doit produire ses unités à l'avance. De même, elle peut avoir intérêt à produire plus lorsque le coût de production est plus faible.

Mais produire en avance conduit à devoir stocker les produits en attendant leur livraison. Un produit non livré en fin de semaine $i-1$ coûte t_i lorsqu'il est stocké pendant la semaine i.

Le but de l'entreprise est de plannifier sa production de façon à satisfaire la demande chaque semaine (le nombre d'unités en stock en fin de semaine $i-1$ plus le nombre d'unités produites la semaine i doit être au moins égale à D_i) tout en minimisant la somme des coûts de production et des coûts de stockage.

On suppose qu'au départ le stock est nul, et on souhaite que le stock soit également nul en fin de semaine H.

Afin de définir un premier modèle de ce problème, on en définit les variables : x_i est le nombre d'unités produites dans la semaine i. s_i est le stock restant en fin de semaine i après avoir satisfait la demande.

$$
\begin{array}{lll}
\min & \sum_{i=1}^{H} c_i x_i + t_i s_{i-1} & \\
s_0 = 0 & & \\
s_H = 0 & & \\
\forall i \in \{1,\dots,H\} & s_i & = x_i + s_{i-1} - D_i \\
\forall i \in \{1,\dots,H\} & x_i & \leq Q_i \\
\forall i \in \{1,\dots,H\}, & x_i, s_i & \in \mathbb{N}
\end{array}
$$

Si l'on considère maintenant que la phase k correspond à la décision de production pour la semaine k, pour plannifier sur les semaines $k,\dots,H$, on a juste besoin de connaître l'état du stock en fin de semaine $k-1$, peu importe les quantités produites auparavant. Le stock va donc jouer le rôle d'état dans le schéma de programmation dynamique.

Ainsi, la valeur de x_k correspond à la décision de la phase k.

Supposons que l'on ait en fin de phase $k-1$ un stock S, et que l'on cherche à déterminer les décisions possibles à la phase k.

On sait que $x_k \leq Q_k$, et que la demande doit être satisfaite à la fin de la

semaine k : $s + x_k \geq D_k$. On a donc

$$\mathcal{D}_k = \{\delta^x | \quad \max(0, D_k - s) \leq x \leq Q_k\}$$

Remarquons que cet ensemble peut être vide.

Si s est l'état en fin de phase $k-1$, et qu'une décision δ^x est prise, l'état résultant (stock en fin de semaine k) est alors :

$$\delta^x | s = s + x - D_k$$

Le coût immédiat $c(\delta^x, s, k) = c_k.x + t_k.s$ (coût de production et de stockage de la semaine k).

L'objectif est alors la somme des coûts immédiats.

Nous avons maintenant tous les éléments pour établir l'équation de récurrence arrière :

$$F_k(s) = \min_{\max(0, D_k - s) \leq x \leq Q_k} (c_k.x + t_k.s + F_{k+1}(s + x - D_k))$$

Si dans cette équation, l'ensemble $\{x| \quad \max(0, D_k - s) \leq x \leq Q_k\} = \emptyset$, par convention $F_k(s) = +\infty$ (le min des valeurs d'un ensemble vide est $+\infty$).

On peut également calculer la valeur initiale, sachant que le stock final doit être nul :

$$F_H(s) = \begin{array}{|ll} +\infty & si \quad s > D_H \ ou \ s + Q_H < D_H \\ c_H(D_H - s) + t_H s & sinon \end{array}$$

Afin d'écrire un algorithme, il reste à déterminer l'espace de variation des états. Pour cela il suffit de disposer d'un majorant de la valeur du stock. On peut par exemple prendre la somme des demandes, ou la somme des capacités de production.

Posons $S = \min(\sum_{k=1}^{H} D_k, \sum_{k=1}^{H} Q_k)$.

L'algorithme 7 implémente le schéma de programmation dynamique ci-dessus. La valeur $+\infty$ doit évidemment être interprétée en termes informatiques, soit par une valeur très très grande, soit comme une exception à gérer dans les calculs de min.

La complexité de cet algorithme est $O(HSQ_{max})$ où $Q_{max} = \max_{k \in \{1,\ldots,H\}} Q_k$.

3.4 Ordonnancement de profit maximal

On considère un ensemble de n tâches devant être effectuées séquentiellement par une machine. Chaque tâche est caractérisée par une durée p_i, une date d'échéance d_i et un profit w_i. Un ordonnancement consiste à attribuer un date d'exécution t_i à chaque tâche i, de sorte que la machine ne fasse qu'une tâche à la fois. Une tâche est alors dite à l'heure si $t_i + p_i \leq d_i$ (si elle se termine au plus tard à son échéance). Une tâche i qui est à l'heure rapporte un profit w_i, lorsqu'elle est en retard elle ne rapporte rien.

Algorithm 7 Gestion de stock

```
pour s = 1 à S faire
    si s > D[H] ou s + Q[H] < D[H] alors
        F[H][s] = +∞
    sinon
        F[H][s] = c[H](D[H] − s) + t[H]s

pour k = H − 1 à 1 faire
    pour s = 1 à S faire
        F[k][s] = +∞
        pour x = max(0, D[k] − s) à Q_k faire
            si F[k][s] > c[k].x + t[k].s + F[k + 1][s + x − D[k]] alors
                F[k][s] = c[k].x + t[k].s + F[k + 1][s + x − D[k]]

opt = F[1][0]
si opt < +∞ alors
    s = 0
    pour k = 1 à H − 1 faire
        y = max(0, D[k] − s)
        tant que F[k][s] ≠ c[k].y + t[k].s + F[k + 1][s + y − D[k]] faire
            y = y + 1
        s = s + y − D[k]  x[k] = y
    x[H] = D[H] − s
```

Retourner x et *opt*

On cherche donc à définir un ordonnancement des tâches de profit maximal (somme des profits des tâches à l'heure).

Afin de définir un schéma de programmation dynamique pour ce problème, on doit tout d'abord étudier les propriétés que peuvent avoir les ordonnancements optimaux, afin de réduire l'espace de recherche.

Définition 3.1. *On dit qu'un sous-ensemble S de solutions d'un problème d'optimisation est* **dominant** *s'il existe une solution optimale du problème qui appartient à S.*

NB : cela ne veut pas dire qu'il n'existe pas d'autre solution optimale.

Lemme 3.2.

Lemme 3.3. *L'ensemble des ordonnancements pour lesquels toutes les tâches à l'heure sont exécutées avant les tâches en retard est dominant.*

Pour prouver cette propriété, on suppose que l'on a un ordonnancement optimal où une tâche en retard, i, est placée juste avant une tâche à l'heure j. On montre qu'en repoussant j à la fin de l'ordonnancement, et en avançant i de p_j, on obtient un ordonnancement où les tâches à l'heure restent à l'heure. En itérant cette transformation, on obtient un ordonnancement optimal.

Lemme 3.4. *L'ensemble des ordonnancements pour lesquels les tâches à l'heure exécutées avant les tâches en retard sont faites par ordre d'échéances croissante est dominant.*

Supposons qu'on ait un ordonnancement optimal où les tâches à l'heure sont avant les tâches en retard. Supposons que dans cet ordonnancement, on ait deux tâches consécutives i et j, avec $d_i > d_j$ qui sont à l'heure. En échangeant la position de ces tâches, elles restent à l'heure, et donc le profit de l'ordonnancement reste identique. En itérant cette transformation on obtient un ordonnancement où les tâches à l'heure sont par ordre d'échéance croissante.

Supposons que les tâches soit numérotées de sorte que $d_1 \leq \ldots \leq d_n$.

Définition 3.5. *On appelle séquence à l'heure un sous-ensemble $H = \{i_1, \ldots, i_k\}$ de tâches, $i_1 \leq \ldots \leq_i k$, telles que, mises dans l'ordre, elles sont à l'heure : On a*

$$\forall j \in \{1, \ldots, k\} \quad t_{i_j} + p_{i_j} = \sum_{x=1}^{j} p_{i_x} \leq d_{i_j}$$

Le profit d'une séquence à l'heure est

$$W(H) = \sum_{j \in \{1,\ldots,k\}} w_{i_j}.$$

Les résultats de dominance nous montrent qu'il existe un ordonnancement optimal dont les tâches à l'heure forment une séquence à l'heure. On est donc ramené au problème de trouver une séquence à l'heure de profit maximal.

On va considérer un processus de décision permettant de construire une séquence à l'heure : à la phase k, on décide ou non d'ajouter la tâche k à la séquence à l'heure en cours.

Pour savoir quelles décisions prendre à partir de la phase k, on a besoin uniquement de connaître la date de fin des tâches déjà ordonnancées dans la séquence précédente. C'est donc cette durée qui va jouer le rôle d'état.

Partant de l'état T au début de la phase k, les décisions possibles sont : si $T + p_k > d_k$, la tâche k ne peut pas être incluse dans une séquence à l'heure. La seule décision possible est donc δ_k^0 (on ne prend pas la tâche k). Dans le cas contraire, on a deux décisions possibles : δ_k^0, δ_k^1. Les transitions sont les suivantes :

$$\delta_k^0 | T = T, \quad \delta_k^1 | T = T + p_k$$

Le profit immédiat de δ_k^0 est 0, celui de δ_k^1 est w_k.

Les valeurs possibles de l'état, à chaque phase sont $\{0, \ldots, \sum_{i=1}^{n} p_i\}$.

On a donc tous les éléments pour définir un schéma de programmation dynamique pour résoudre ce problème :

$$F_n(T) = \begin{cases} w_n & si \quad T + p_n \leq d_n \\ 0 & sinon \end{cases}$$

$$F_k(T) = \begin{cases} F_{k+1}(T) & si \quad T + p_k \leq d_k \\ \max(F_{k+1}(T), w_k + F_{k+1}(T - p_k)) & sinon \end{cases}$$

3.5 Voyageur de commerce

On considère n villes et une matrice D de distances entre ces villes. Un tour est un ordre de visite des villes, partant de la ville 1 sans jamais repasser par une ville déjà visitée. La longueur d'un tour est la somme des distances entre les villes successives, et l'on recherche un tour de longueur minimale.

On peut considérer un processus de décision pour construire un tour qui consiste à choisir d'abord la ville visitée en second, puis en second,..., puis en n-ième position.

Mais dans ce cas, lorsqu'on en est à la phase k, on a besoin, pour construire la suite des décisions de connaître l'ensemble des villes déjà visitées (pour ne pas y repasser) et la ville en position $k-1$. Ainsi, l'espace des états $\mathcal{E}_k$ est le produit cartésien de l'ensemble des sous-ensembles de k villes et de l'ensemble des villes.

Les décisions qu'on peut prendre à partir d'un état $(E, v) \in \mathcal{E}_{k-1}$ sont de la forme δ_k^j pour chaque ville $j \notin E$. Leur coût immédiat est $D(v, j)$. Les transitions sont de la forme $\delta_k^j|(E, v) = (E \cup \{j\}, j)$.

On peut donc trouver un schéma de programmation dynamique : Pour ce qui concerne $F_n(E, v) = D(v, j)$ où j est la seule ville qui n'appartient pas à E.

$$F_k(E, v) = \min_{j \notin E} D(v, j) + F_{k+1}(E \cup \{j\}, j)$$

Le problème de ce schéma, c'est que le nombre d'états est exponentiel. En effet, le cardinal de $\mathcal{E}_k$ est $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Par conséquent, l'algorithme de résolution ne peut pas être efficace, et est même inabordable pour un nombre de villes même assez petit.