Chapitre 5 : Programmation dynamique

5.1 Introduction

La programmation dynamique (DP) est une technique algorithmique pour résoudre un problème d'optimisation en le décomposant en sous-problèmes plus simples et en utilisant le fait que la solution optimale au problème global dépend de la solution optimale à ses sous-problèmes. La programmation dynamique, proposée par Bellman (1957), est basée sur le principe d'optimalité de Bellman : *" Toute sous politique, d'une politique optimale, est optimale"*.

Contrairement à la programmation linéaire, il n'y a pas un formalisme mathématique standard. C'est une approche de résolution où les équations doivent être spécifiées selon le problème à résoudre.

5.2 Programmation dynamique versus diviser et régner

Il y a une grande ressemblance entre la programmation dynamique et la méthode diviser et régner en ce sens que, la solution finale d'un problème dépend des solutions précédentes obtenues pour ces sous-problèmes. Cependant, la différence significative entre ces deux méthodes réside dans la structure des sous-problèmes. Dans la programmation dynamique les sous solutions peuvent etres superposés. Tandis que dans l'approche diviser et régner, les sous-problèmes sont complètement séparés et peuvent être résolus indépendamment l'un de l'autre. (figure 5.1).

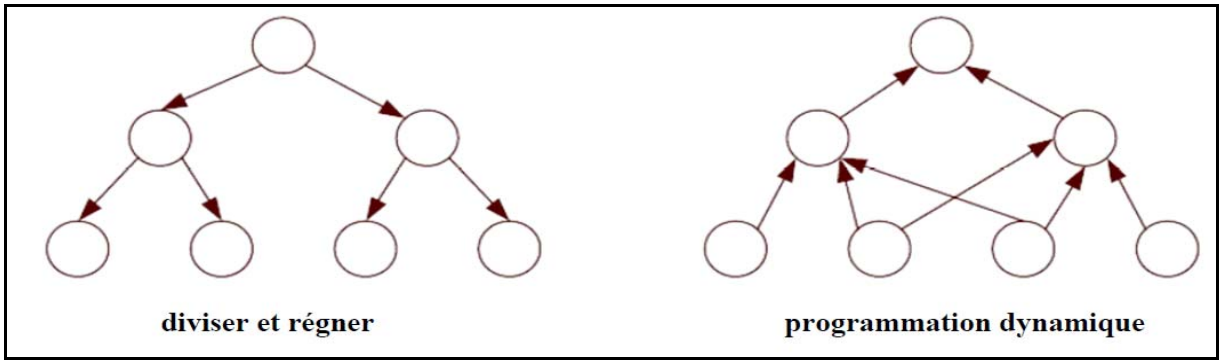


Figure 5.1 – différence entre la programmation dynamique et la méthode « diviser pour régner »

Une seconde différence entre ces deux méthodes est, comme illustré par la figure ci-dessus, est que la méthode diviser et régner est récursive, les calculs se font de haut en bas. Tandis que la programmation dynamique est une méthode dont les calculs se font de bas en haut : on commence par résoudre les plus petits sous-problèmes. En combinant leur solution, on obtient les solutions des sous-problèmes de plus en plus grands.

5.3 Paradigme de la programmation dynamique

La programmation dynamique est une technique simple de conception d'algorithmes exacts qu'on peut appliquer pour résoudre un problème si:

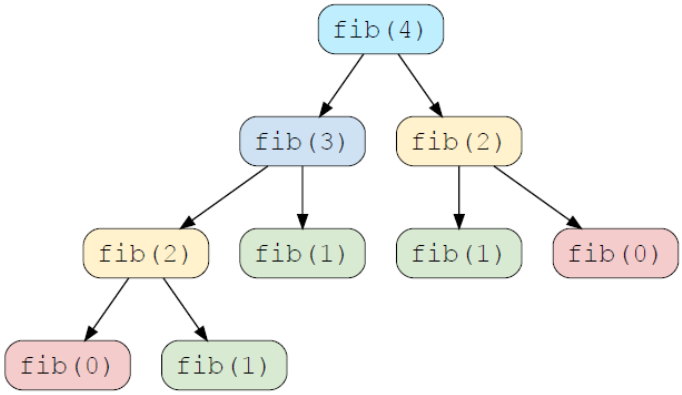
1**.** La solution optimale d'un problème de taille *n* s'exprime en fonction de la solution optimale des sous-problèmes de taille inférieure à *n* (le principe d'optimalité).

2. Dans la version récursive les mêmes calculs sont répétés plusieurs fois comme montre l'exemple suivant de la fonction fibonacci .

Comme nous le savons tous, les nombres de Fibonacci sont une série de nombres dans lesquels chaque nombre est la somme des deux nombres précédents. Les premiers nombres de Fibonacci sont 0, 1, 1, 2, 3, 5 et 8, et ils continuent à partir de là. D'une manière génarlen la série de Fibonacci est donnée par la formule suivante

Fib(1)=Fib(0)=1

Fib(n) = Fib(n-1) + Fib(n-2), pour n > 1



Nous pouvons clairement voir le modèle de sous-problème qui se chevauchent ici, car fib (2) a été évalué deux fois et fib (1) a été évalué trois fois.

Solution? L'idée est alors simplement d'éviter de calculer plusieurs fois la solution du même sous-problème.

Pour cela, on définit une table pour mémoriser les calculs déjà effectués : à chaque élément correspondra la solution d'un et d'un seul problème intermédiaire. La programmation dynamique offre deux approches

5.3.1 De haut en bas avec mémorisation (Top-down with Memoization)

Dans cette approche, nous essayons de résoudre le plus gros problème en trouvant récursivement la solution à des sous-problèmes plus petits. Chaque fois que nous résolvons un sous-problème, nous mettons son résultat en cache afin de ne pas le résoudre à plusieurs reprises s'il est appelé plusieurs fois. Au lieu de cela, nous pouvons simplement renvoyer le résultat enregistré. Cette technique de stockage des résultats de sous-problèmes déjà résolus et appelée mémorisation.

Nous verrons cette technique dans notre exemple de nombres de Fibonacci. Voyons d'abord la solution récursive pour trouver le nième nombre de Fibonacci:

```
public int CalculateFibonacci(int n) {
    if(n < 2)
      return n;
    return CalculateFibonacci(n-1) + CalculateFibonacci(n-2);
  }
```

Comme nous l'avons vu ci-dessus, ce problème montre le modèle de sous-problèmes qui se chevauchent, alors utilisons la mémorisation ici. Nous pouvons utiliser un tableau pour stocker les sous-problèmes déjà résolus.

```
public int CalculateFibonacci(int n) {
    int memoize[] = new int[n+1];
    return CalculateFibonacciRecursive(memoize, n);
  }
  public int CalculateFibonacciRecursive(int[] memoize, int n) {
    if(n < 2)
      return n;
    // if we have already solved this subproblem, simply return the result fro
m the cache
    if(memoize[n] != 0)
      return memoize[n];
    memoize[n] = CalculateFibonacciRecursive(memoize, n-
1) + CalculateFibonacciRecursive(memoize, n-2);
  return memoize[n];
  }
```

5.3.2 Version itérative (de bas en haut):

La tabulation est l'opposé de l'approche descendante et évite la récursivité. Dans cette approche, nous résolvons le problème «de bas en haut» (c'est-à-dire en résolvant d'abord tous les sous-problèmes connexes). Cela se fait généralement en remplissant une table à n dimensions. Sur la base des résultats du tableau, la solution au problème principal / d'origine est ensuite calculée.

La tabulation est l'opposé de la mémorisation, car dans la mémorisation, nous résolvons le problème et maintenons une carte des sous-problèmes déjà résolus. En d'autres termes, dans la mémorisation, nous le faisons de haut en bas dans le sens où nous résolvons d'abord le problème principal (qui revient généralement vers le bas pour résoudre les sous-problèmes).

Appliquons la tabulation à notre exemple de nombres de Fibonacci. Puisque nous savons que chaque nombre de Fibonacci est la somme des deux nombres précédents, nous pouvons utiliser ce fait pour remplir notre table. Voici le code de notre approche de programmation dynamique ascendante:

```
public int CalculateFibonacci(int n) {
    if (n==0) return 0;
    int dp[] = new int[n+1];

    //base cases
    dp[0] = 0;
    dp[1] = 1;

    for(int i=2; i<=n; i++)
      dp[i] = dp[i-1] + dp[i-2];

    return dp[n];
  }
```

5.3.3 Exemple prototype. Le problème du voyageur

Le problème de voyageur est un exemple typique de la programmation dynamique. Le voaygeur décide d'effectuer un voyage entre deux villes passant par un ensemble de villes. Son but est de choisir le chemin le moins couteux en termes de couts. Les villes sont organisées selon un graphe à étage multiple. Un graphe à plusieurs étages est un graphe orienté valué dans lequel les nœuds peuvent être divisés en un ensemble d'étapes de telle sorte que toutes les arêtes soient d'une étape à une étape suivante uniquement (en d'autres termes, il n'y a pas d'arête entre les sommets de la même étape et d'un sommet de courant étape à l'étape précédente (voir la figure ci-dessous).

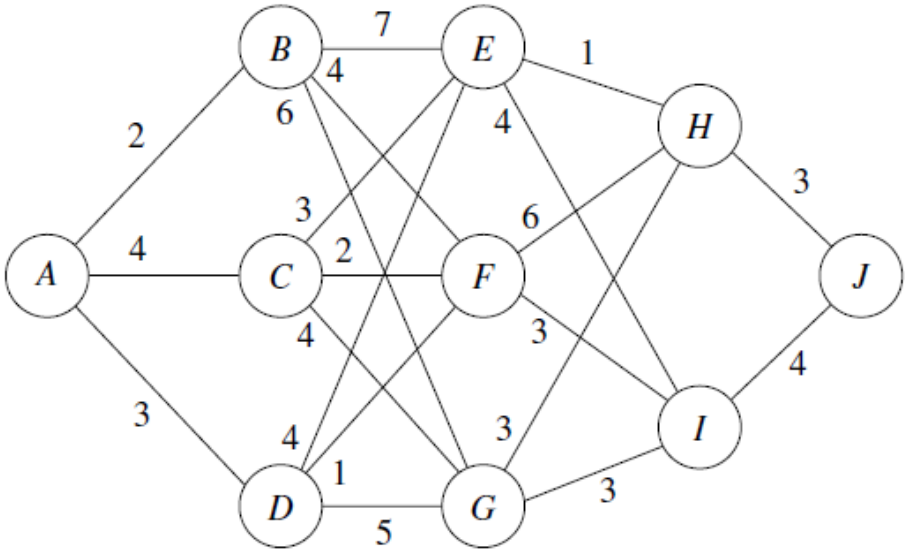


Figure 5.2 – exemple du problème de voyageur.

Dans cette exemple, le chemin du voyageur est composé de 4 étapes (figure 5.2). Le voyageur de démarre de la ville A et il doit arriver à la ville J. Les arcs entre ces nœuds représentent les différents trajets possibles et les valeurs c_{ij} représentent le coût relatif à l'arc.

Exemple : Si le voyageur empreinte le trajet $A \rightarrow C \rightarrow F \rightarrow I \rightarrow J$, le coût total du voyage est 13.

Résolution par la programmation dynamique

Pour résoudre ce problème à l'aide de la programmation dynamique, il faut définir un ensemble de paramètres :

✓ ***Définir les étapes du problème.***

Dans ce graphe, il y a 4 étapes

✓ ***Définir les variables de décision.***

Soit x_n *(n=1, ..., 4)* les variables de décisions relatives à chacune des 4 étapes. Chaque variable x_n

Peut prendre les nœuds d'une seule étape. Exemple pour x_1, les valeurs possibles sont B, C ou E, alors que x_4 prendre que la valeur J.

Définir une fonction récursive pour calculer le cout de la solution .

Soit $f_n(S, x_n)$ le coût total pour le reste des étapes sachant que nous somme à l'état S de l'étape *n* et que la destination choisie est x_n.

Etant donné *S* et *n*, soit x_n^* la valeur de x_n qui minimise $f_n(S, x_n)$ et soit $f_n^*(S)$ la valeur minimale de $f_n(S, x_n)$. donc

$f\,n^*(s) = \min_{xn} fn(s, xn) = fn(s, xn^*)$

où

$f_n(s, x_n)$ = cout actuel (étape n) + minimum future cout (étape $n+1$ en avant

$= c_{sx} + f_{n+1}^*(x_n)$.

D'après la formule précédente, la démarche de la programmation dynamique commence d'abord par les sous problèmes qui se situent chronologiquement les derniers et sur un principe de retour en arrière.

La valeur de c_{sx} est donnée par les tableaux précédents pour c_{ij} en mettant $i = s$ (l'étape actuelle) et $j = x_n$ (la destination immédiate. Parce que la destination finale (étape J) est atteinte à la fin de l'étape 4, donc lecout de l'étape 5 $f_5^*(J) = 0$.

L'objectif est donc de trouver la valeur de $f_1^*(A)$ et le chemin correspondant. la programmation dynamique trouve la par trouver successivement les valeurs de $f_4^*(s)$, $f_3^*(s)$, $f_2^*(s)$, pour chaque étape possible s et ensuite en utilisant $f_2^*(s)$ pour resoudre $f_1^*(A)$.

La procédure de résolution

On va commencer par la dernière étape

Etape 4

Ici le voyageur a une seule étape à passer (n=4), le chemin passe par les états actuels H ou I

S \ x_4	$f_4(S, x_4))$	$f_4^*(s)$,	x_4^*
H	*3*	*3*	*J*
I	*4*	*4*	*J*

Etape 3

	$f_3(S, x_3)=C_s x_3 + f_4^*(x_3)$			
S \ x_3	*H*	*I*	$f_3^*(s)$,	x_3^*
E	*4(=1+3)*	*8(=4+4)*	*4*	*H*
F	*9*	*7*	*7*	*I*
G	*6*	*7*	*6*	*H*

Etape 2

	$f_2(S, x_2)=C_s x_2 + f_3^*(x_2)$				
S \ x_2	E	F	G	$f_2^*(s)$	x_1^*
B	11	11	12	11	E ou F
C	7	9	10	7	E
D	8	8	11	8	E ou F

Etape 1

	$F_1(S, x_1)=C_s x_1 + f_2^*(x_1)$				
S \ x_1	B	C	D	$f_1^*(1)$	x_1^*
A	13	11	11	11	C ou D

La valeur de $f_1^*(A) = 11$ représente le coût total pour le voyage. Le chemin optimal n'est unique puisque dès le départ on peut choisir $x_1^* = 3$ ou 4, donc l'ensemble de ces chemins est:

$1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 10$

$1 \rightarrow 4 \rightarrow 5 \rightarrow 8 \rightarrow 10$

$1 \rightarrow 4 \rightarrow 6 \rightarrow 9 \rightarrow 10$

5.4 Caractéristiques d'un problème de programmation dynamique

Nous allons maintenant sur la base de l'exemple précédant analyser les propriétés communes aux problèmes de programmation dynamique.

- ✓ Le problème peut être décomposé en étapes et une décision doit être prise à chaque étape.

  L'exemple du voyageur est devisé en 4 étapes où à chaque étape. La décision politique à chaque étape était de choisir la prochaine destination

- ✓ A chaque étape correspond un certain nombre d'états. Les états sont les différentes conditions possibles dans lesquelles le système pourrait être à une étape du problème. Le nombre d'états peut être finie ou infini.

- ✓ A chaque étape, la décision prise transforme l'état actuel en un état associé à l'étape suivante. Dans cet exemple, se trouvant à une ville donnée, le voyageur décide de se rendre à une autre ville qui est un état de l'étape suivante.

- ✓ Etant donné un état, une stratégie optimale pour les étapes restantes est indépendante des décisions prises aux étapes antécédentes.
- ✓ La procédure de recherche de la solution optimale commence par trouver la décision optimale pour la dernière étape.
- ✓ Une relation de récurrence qui identifie la stratégie optimale de l'étape n va trouver la stratégie optimale d pour l'étape n+1.

Dans notre exemple la relation est $f_n^*(S) = \underset{x_n}{Min}\{C_S x_n + f_{n+1}^*(x_n)\}$

La stratégie optimale étant donné que nous sommes à l'état *S* de l'étape *n*, nécessite de retrouver la valeur de x_n qui minimise l'expression ci-dessus.

La relation de récurrence à toujours cette forme $f_n^*(S) = \underset{x_n}{Max}$ *ou* $\underset{x_n}{Min}\{f_n(S,x_n)\}$,

La forme précise de la relation récursive diffère d'un problème à un autre s. Cependant, une notation analogue à celle introduite dans l'exemple précèdent est à utiliser, comme résumé ci-dessous la notation utilisée :

- n = etiquette de l'étape courante ($n = 1, 2, \ldots, N$).
- s_n = état actuel de l'étape n.
- x_n = variable de decision de l'étape n. x_n^* = valeur optimale de x_n (sachant s_n).
- $f_n(s_n, x_n)$ = contribution des étapes $n, n+1, \ldots, N$ dans la function objectif si le système démarre de l'état s_n à l'étape n, la décision immediate est x_n, et des décisions optimales sont prises par la suite.
  $f_n^*(s_n) = f_n(s_n, x_n^*)$.
- La relation récursive sera toujours de la forme

$$f_n^*(s_n) = \max_{x_n}\{f_n(s_n, x_n)\} \quad \text{ou} \quad f_n^*(s_n) = \min_{x_n}\{f_n(s_n, x_n)\},$$

- ✓ Utilisant cette relation de récurrence, l'algorithme procède étape par étape en commençant par la dernière étape et en reculant vers la première étape. Dans tout problème de programmation dynamique, on peut construire à chaque étape un tableau analogue au suivant.

Etape n

	$f_n(S, x_1)$		
S \ X_1	*états n+1*	$f_n^*(s)$	x_n^*
états n	*le coût ou la distance*	*La longueur du chemin optimal à partir de S jusqu'à l'état final*	*Le meilleur état de l'étape n+1 le long du chemin optimal final*

- ✓ **Lecture de la solution** : l'étape finale ne conduit qu'à la valeur (optimale) du problème de départ. Elle ne donne pas directement la solution conduisant à cette valeur. En générale, pour avoir cette solution, on fait un travail inverse en lisant dans la table en partant de la solution finale et en faisant le chemin inverse des calculs effectués à partir de l'étape 1.

Exemple 5.3 : distribution de nouveaux chercheurs sur des équipes.

Un projet spatial gouvernemental mène des recherches sur un certain problème d'ingénierie qui doit être résolu avant que les gens puissent voler en toute sécurité vers Mars. Trois équipes de recherche essaient actuellement trois approches différentes pour résoudre ce problème. Il a été estimé que, dans les circonstances actuelles, la probabilité que les équipes respectives - 1, 2 et 3 - ne réussissent pas est de 0,40, 0,60 et 0,80, respectivement. Ainsi, la probabilité actuelle d'échec des trois équipes est de (0,40) (0,60) (0,80) = 0,192. Étant donné que l'objectif est de minimiser la probabilité d'échec, deux autres scientifiques de haut niveau ont été affectés au projet. Le tableau suivant donne la probabilité estimée que les équipes respectives échouent lorsque 0, 1 ou 2 scientifiques supplémentaires sont ajoutés à cette équipe. Seuls les nombres entiers de scientifiques sont pris en compte car chaque nouveau scientifique devra consacrer toute son attention à une équipe. Le problème est de déterminer comment affecter les deux scientifiques supplémentaires afin de minimiser la probabilité que les trois équipes échouent.

	Probabilité d'échec		
Nbre de nouveaux chercheurs	*Groupes*		
	1	*2*	*3*
0	*0,4*	*0,6*	*0,8*
1	*0,2*	*0,4*	*0,5*
2	*0,15*	*0,2*	*0,3*

Solution: nous commençonpar la modelisation du problème.

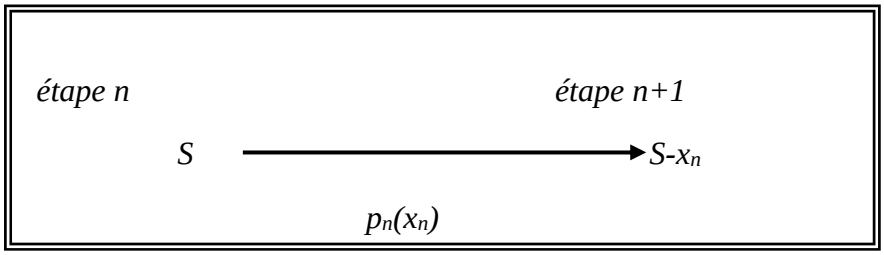
- **Etapes:** 3 étapes ou les états représentent le nombre de chercheurs disponibles (S)
- **Variable de décision:** x_n représente le nombre de chercheurs à allouer à l'équipe de recherche *n, n = 1, 2, 3.*

Contribution de la décision x_n est la probabilité que l'équipe n échoue après avoir eu x_n chercheur de plus.

- La relation récursive $f_n^*(S)$ est la probabilité minimale que les groupes *n=1,2, 3* échouent dans leurs recherches :

$f_n^*(S) = \underset{x_n \le S}{Min}\, f_n(S, x_n)$ $n = 1, 2, 3$

avec $f_n(S, x_n) = p_n(x_n) \times f_{n+1}^*(S\text{-}xn)$, $p_n(x_n)$ est la contribution de la décision x_n et $f_3^*(s) = 1$.



$f_n(S, x_n) = p_n(x_n)\ f_{n+1}^*(S - x_n)$

Dans cette exemple la relation récursive n'est pas additive dans le temps, mais c'est une multiplication.

Etape 3

	$f_3(S, x_3) = p_3(x_3)$				
S \ x_3	0	1	2	$f_3^*(S)$	x_3^*
0	0,8	-	-	0,8	0
1	0,8	0,5	-	0,5	1
2	0,8	0,5	0,3	0,3	2

Etape 2

	$f_2(S, x_2) = p_2(x_2)\ f_3^*(x_3)$				
S \ x_2	0	1	2	$f_2^*(S)$	x_2^*

0	*0,48*	-	-	*0,48*	*0*
1	*0,3*	*0,32*	-	*0,3*	*0*
2	*0,18*	*0,2*	*0,16*	*0,16*	*2*

Etape 1

	$f_1(S, x_1) = p_1(x_1)\ f_2^*(x_1)$				
S \ x_1	*0*	*1*	*2*	$f_1^*(S)$	x_1^*
2	*0,064*	*0,06*	*0,072*	*0,06*	*1*

La stratégie optimale est $x_1^* = 1$, $x_2^* = 0$ et $x_3^* = 1$.

La probabilité d'échec des trois groupes de recherche est de 0,06.

Exercice à faire : résoudre le *problème de sac à dos* en utilisant la programmation dynamique

Référence :

R. Andreani, "Chapter 11: Dynamic Programming," 2009

E.V. Denardo. Dynamic Programming. Prentice-Hall, Eglewood Cliffs, 2003.