International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

Optimisation De L'analyse De La Valeur Par L'utilisation De La Programmation Dynamique

[Optimization Of Value Analysis Through The Use Of Dynamic Programming]

Harimbinintsoa RAVAOMIALITIANA[1], Huchard Paul Berthin RANDRIANIRAINY[2], Jaconnet Oliva ANDRIANAIVORAVELONA[3]

[1]Centre National de Recherches Industrielle et Technologique
Antananarivo, Madagascar
[2]Centre National de Recherches Industrielle et Technologique
Antananarivo, Madagascar
[3]Ecole Supérieure Polytechnique, Université d'Antananarivo Madagascar
Auteur correspondant : Harimbinintsoa RAVAOMIALITIANA. E-mail : ravaomialitiana@gmail.com

Résumé : Dans un contexte marqué par la nécessité d'une gestion optimisée des coûts et de la valeur dans les projets de construction, l'analyse de la valeur (AV) demeure un outil stratégique. Toutefois, sa mise en œuvre se limite souvent à des approches qualitatives ou semi-quantitatives, rendant complexe l'identification des meilleures alternatives lorsqu'interviennent des contraintes budgétaires. Cette recherche propose une intégration de la programmation dynamique dans la démarche d'AV, appliquée au cas d'un bâtiment universitaire à Madagascar. L'objectif est de développer un modèle quantitatif permettant d'optimiser les choix fonctionnels en fonction des coûts et des valeurs associées. En s'appuyant sur la théorie de Bellman et le principe de décomposition en sous-problèmes, un algorithme inspiré du problème du sac à dos a été adapté. Celui-ci permet de rechercher soit une optimisation maximale, soit une optimisation minimale, selon les besoins de la décision. Les résultats obtenus mettent en évidence une structure algorithmique capable d'améliorer le processus de sélection des fonctions dans l'AV, en renforçant la rationalité et la précision des décisions. Cette approche contribue à enrichir la méthodologie traditionnelle de l'AV et ouvre des perspectives pour une gestion plus efficiente des projets de construction.

Mots Clés : Analyse de la valeur (AV), programmation dynamique, optimisation, coûts, construction, Décision multicritère

Abstract: In a context marked by the need for optimized cost and value management in construction projects, Value Analysis (VA) remains a strategic tool. However, its implementation is often limited to qualitative or semi-quantitative approaches, making it difficult to identify the best alternatives when budget constraints are involved.
This research proposes the integration of dynamic programming into the VA process, applied to the case of a university building in Madagascar. The objective is to develop a quantitative model capable of optimizing functional choices according to the associated costs and values. Based on Bellman's theory and the principle of problem decomposition, an algorithm inspired by the knapsack problem was adapted. This allows for either maximum or minimum optimization, depending on decision-making needs.
The results highlight an algorithmic structure that improves the selection process of functions in VA, strengthening both the rationality and accuracy of decisions. This approach contributes to enriching the traditional VA methodology and opens perspectives for more efficient management of construction projects.

Keywords: Value Analysis (VA), dynamic programming, optimization, cost, construction, multi-criteria decision

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

1. INTRODUCTION

L'apport de contribution sur le traitement de l'analyse de la valeur avec notre approche et notre méthode pour pouvoir optimiser la qualité de l'optimisation. Dans un premier temps, nous allons considérer la présentation de la liaison de notre thème avec la solution à apporter ensuite la contribution sur la façon de choisir de solution optimale par intermédiaire de l'algorithme basée sur la programmation dynamique

La répartition du bloc du problème est en sous problème va justifier la théorie de Bellman en programmation dynamique : « une politique est optimale si Toutes les sous politique d'une politique soit optimale ». En plus les sous-solutions trouvées peuvent être utilisées pour pouvoir optimiser les autres sous-politiques qui ne sont pas encore optimisées. La technique est avantageuse par rapport aux autres algorithmes comme « diviser pour régner » qui traiter un problème en sous plusieurs sous problèmes indépendants entre eux et sans interactions ou sans entraide.

2. METHODOLOGIES

2.1 Démarche générale en programmation dynamique.

La programmation dynamique (dynamic programming ou encore dynamic optimization en anglais) est une technique d'optimisation d'un algorithme visant à éviter de recalculer des sous-problèmes en stockant les résultats en mémoire. L'idée est simple, mais le gain sur la complexité en temps peut être considérable, et cette technique est très largement utilisée dans de nombreux algorithmes. Cette technique de programmation suit le principe d'optimalité de Bellman énonçant que la solution optimale d'un problème peut être calculée à partir de solutions optimales de sous-problèmes

2.1.1 Programmation dynamique et l'analyse de la valeur

Comme l'analyse de la valeur est l'étude d'un ensemble de processus décomposable en plusieurs phases, alors pour chaque phase il est préférable de traiter le moyen d'avoir une valeur optimale.

Dans ce cas il existe une suite de décision d_1, d_2, …, dk_{-1}, d_k, dk_{+1}…, dn où dn est la décision finale à optimiser mais aussi un résultat des suites successives de décisions d_i ont chacune est optimisée.

L'objectif est alors d'appliquer la technique de la programmation dynamique dans les fonctions de coûts dont l'orientation est la minimisation.

2.1.2 Processus de décision décomposable en phases

Dans le cadre de la pratique de l'Analyse de la valeur, le système en entier est décomposable en phase. A la suite de sept(7) étapes principales l'optimisation de la future solution à traiter est essentielle, la plus part des modèles choisit tout simplement un propos sans entrer dans le détails avec les autres éléments qui accompagnent le choisit.

Comme proposition de solution, chaque élément résultant de l'ensemble de l'étape de base fait l'objet de l'optimisation d'où le choix de la programmation dynamique et en plus chaque optimisation obtenue collabore avec autres éléments qui ont besoin de sa participation au moins le résultat de son utilisation.

L'ensemble des points cités ci-dessous précise la démarche associée :

- Un processus de décision dans l'AV se décompose en une suite de N phases de décision.
- Chaque phase k est caractérisée par un ensemble d'états initial Yk-1 et un ensemble d'états terminal Y_k
- Au cours de la phase k, l'évolution d'un état initial y_{k-1} vers un état terminal y_k se réalise par l'intermédiaire d'une décision dont le coût est noté par Vk(yk-1,yk).
- Une politique d (sous forme d'un ensemble) est une succession de décisions d_1 d_2… d_N, permettant de générer une séquence d'états : y_0 y_1 …. y_{N-1} y_N

$$c(d) = \sum_{k=1}^{N} v_k(y_{k-1}, y_k) \quad (1)$$

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

– Le coût d'une telle politique est défini par la solution de deux problèmes qui se posent suivants :

(P1) Etant donné un état initial y_0 et un état terminal y_N , déterminer la politique de coût optimal parmi celles qui évoluent de y_0 à y_N.

(P2) Déterminer une politique de coût optimal parmi celles qui évoluent d'un état donné initial vers l'un des états de y_N

2.1.3 Principe de Bellman

Définition (sous politique) : étant donné une politique d= d_1 d_2… d_N générant la séquence des états : y_0 y_1 …. y_{N-1} y_N. La séquence de décisions d_k=d_1 d_2… d_k qui génère la séquence y_0 y_1…y_k sera dite une sous politique de d.

Principe de Bellman: «Toute sous politique d'une politique optimale est-elle même optimale ».

Ainsi si la politique d= d_1 d_2… d_k… d_N, générant la séquence des états y_0 y_1 ….y_k… y_{N-1} y_N est optimale parmi toutes celles qui permettent d'atteindre y_N à partir de y_0 alors la sous politique d_k= d_1 d_2… d_k est optimale parmi toutes celles qui permettent d'atteindre y_k à partir de y_0.

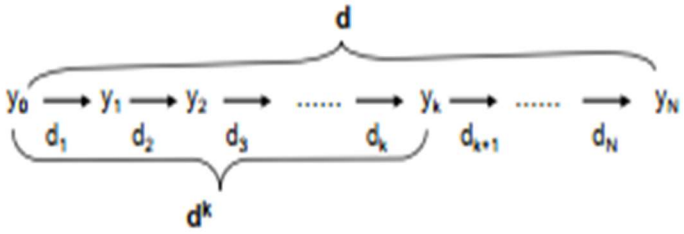

Figure 1 : présentation des décisions intermédiaires.

2.1.4 Formules d'optimisation séquentielles

Comme démarche, le traitement des séquences de décisions est essentiel d'où la présentation de l'optimisation séquentielle est importante. La Figure 02 ci-dessous présente la relation entre les composants de la série de décision.

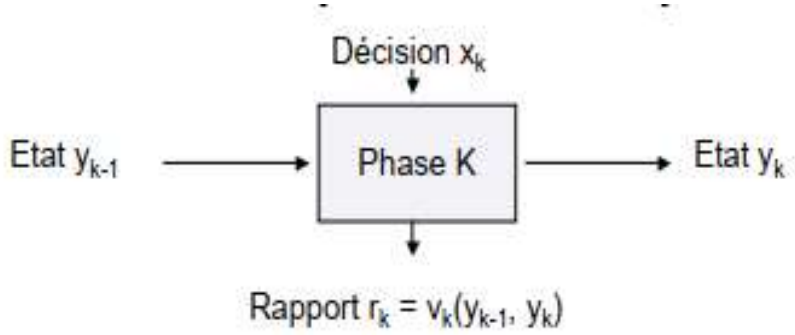

Figure 2 : Optimisation Séquentielle

• On note par $f_k(y_{k-1},y_k)$ le « coût » de la politique minimale permettant d'évoluer de y_0 à y_k (à la fin de l'étape k).

• On a $f_1(Y_0, Y_1) = V_1(Y_0, Y_1)$

• Le principe de Bellman permet de tirer la relation de récurrence suivante :

$$f_k(y_0,y_k)= \min_{y_{k-1}\in D(y_0,y_{k-1})} [f_{k-1}(y_0, y_{k-1}) + V_k(y_{k-1}, y_k)] \quad (2)$$

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

Où $D(y_0,y_{k-1})$ est un sous ensemble de Y_{k-1} défini par :

$y_{k-1} \in D(y_0, y_{k-1})$ ⇔ y_{k-1} est l'avant dernier état d'une politique **d** permettant l'évolution de y_0 à y_k.

2.2 Calcul de la distance par Programmation dynamique

Étant donné 2 séquences **a**=$a_1a_2…a_n$ et **b**=$b_1b_2…b_n$, il est possible de définir un alignement des 2 séquences **a** et **b** par une séquence de décisions consécutives. On note par :

$a^i = a_1a_2…a_i$ la sous séquence de **a** formée des i premiers caractères

$b^j=b_1b_2..b_j$ la sous séquence de b formée des j premiers caractères

Afin de définir le processus décisionnel, on note par $y_{i,j}$ l'état qui représente un alignement particulier des 2 sous séquences a^i et b^j. Étant donné l'état $y_{i,j}$,il est possible de progresser dans l'alignement en prenant l'une des 3 décisions suivantes :

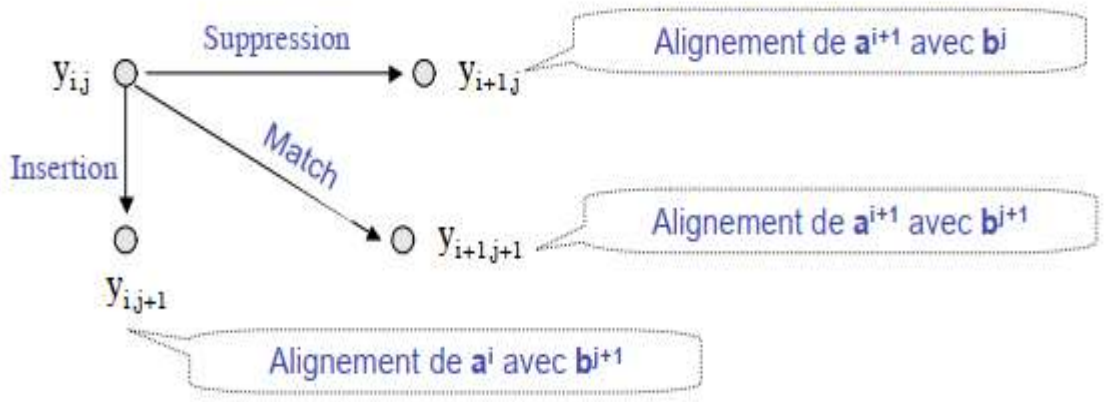

Figure 3 : Progression possible à partir de l'état $y_{i,j}$

2.2.1 Calcul de la distance

Les formules de la programmation dynamique permettent de calculer la distance.

Soit a=$a_1a_2a_3…a_n$ et b=$b_1b_2b_3…b_m$

On définit $D_{i,j}$=D($a_1a_2…a_i$, $b_1b_2…b_j$) où

$$D_{0,0}= 0$$

$$D_{0,j}=\sum_{k=1}^{j} d(-, b_k)$$

$$D_{i,0}=\sum_{k=1}^{i} d(a_k, -)$$

Et la formule de récurrence :

$$D_{i,j} = MIN[D_{i-1,j} + d(a_i,-),D_{i-1,j-1} + d(a_i,b_j),D_{i,j-1} + d(-,b_j)] \quad (3)$$

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

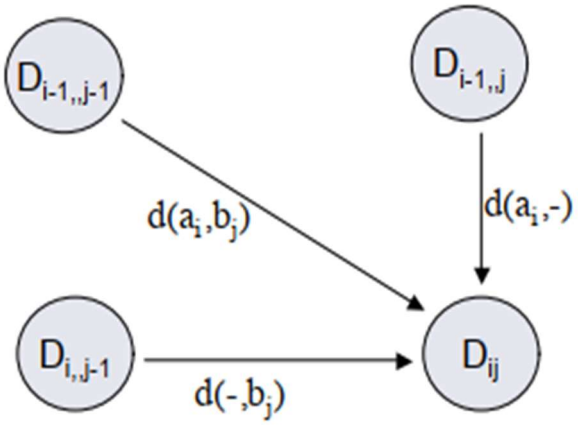

Figure 4 : Présentation du calcul de D_{ij}

2.2.2 Explication de la méthode

La Programmation dynamique utilise des différentes techniques d'optimisation séquentielle et ne s'applique qu'à des problèmes qui peuvent se décomposer en sous-problèmes, dépendant les uns des autres.

Des sous-problèmes ont en commun des « sous-sous-problèmes »

2.2.3 Principe

Un algorithme de Programmation Dynamique résout chaque sous-sous-problème une seule fois et mémorise sa solution dans un tableau, une solution optimale du problème est obtenue à partir de solutions optimales de sous-problèmes. C'est une méthode ascendante, à l'opposer à la récursivité, méthode descendante (diviser pour régner).

Il peut se formuler de la manière suivante : Dans une séquence optimale de décisions, quelle que soit la première décision, les décisions suivantes forment une sous-suite optimale, compte tenu des résultats de la première décision

On appelle souvent Politique la suite des décisions prises, le principe peut aussi s'énoncer : une politique optimale ne peut être formée que de sous-politiques optimales

Une des difficultés essentielles de la programmation dynamique est de reconnaître si un problème donné est justiciable du principe d'optimalité et peut être résolu par cette méthode

Il est impossible de donner des recettes pour répondre à cette question

On peut toutefois remarquer que le principe d'optimalité implique que le problème à étudier puisse être formulé comme celui de l'évolution d'un système

2.2.4 Méthodologie

- Caractériser la structure d'une solution optimale
- Définir par récurrence la valeur d'une solution optimale (mettre en évidence les liens entre les sous problèmes)
- Calculer la valeur d'une solution optimale de manière séquentielle, ascendante
- Construire une solution optimale à partir des informations calculées

A titre de rappel, un problème de combinatoire avec contrainte peut se traduire comme suit :

$\mathbf{min} \sum_{i=1}^{n} x_i p_i$ telle que

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

min$\sum_{i=1}^{n} x_i p_i \leq P$ et $x_i \in \{0,1\}$ (4)

On suppose que $\forall i \ p_i \leq P$ et $\sum_{i=1}^{n} p_i > P$

P est une valeur fixée qu'on ne peut pas dépasser selon l'analyse effectuée pendant la phase de l'analyse fonctionnelle et de la valeur de fonction objective.

La recherche exhaustive, consistant à générer tous les sous-ensembles possibles, a une complexité en temps en **O(n 2^n)** dans le pire cas, et en **O(2^n)** en moyenne.

Pour résoudre une AV par la Programmation Dynamique, il faut faire apparaître des sous-problèmes

Soient les n P sous-problèmes suivants, avec i ∈ [1..n] et j ∈ [1..P]

2.2.5 Utilité maximale

On note U(i, j) l'utilité maximale obtenue avec les i premiers objets et un coût j :

U(i, j) = min $\sum_{k=1}^{i} x_k \ u_k$ telle que

$\sum_{k=1}^{ni} x_k p_k \leq$ j et $x_k \in \{0,1\}$ (5)

La solution est U(n, P) ainsi qu'un contenu optimal donné par le vecteur booléen (x_1,x_2,…,x_n)

On suppose les objets numérotés de 1 à n, tel que $p_1 \leq p_2 \leq \ldots \leq p_n$

Par convention U(i, k) = 0 quand $0 \leq k \leq p_i - 1$

$$U(1,j) = \begin{cases} 0 & \text{si } j < p_1 \\ u_1 & sinon \end{cases} \quad (6)$$

Lorsque $2 \leq i \leq n$ et $p_1 \leq j \leq P$ avec $0 \leq j - p_i$ on a :

U(i, j) = min {U(i-1, j); U(i-1, j-p_i) + u_i } (7)

L'idée de l'algorithme consiste donc à calculer : d'abord les **U(1, j),** quand **$p_1 \leq j \leq P$**

puis : **U(i, j) = min{(U(i-1,j);U(i-1, j-p_i) + u_i},**i variant de 2 à n, où $p_1 \leq j \leq P$ avec $0 \leq j - p_i$

Cela revient donc à remplir le tableau U, de taille n ($P - p_1 + 1$), ligne après ligne et pour chaque colonne de p_1 à P

Extrait de l'algorithme de remplissage du tableau U

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

```
1  for j ∈ [p1..P] do U(1, j) ← u1
2  for i ∈ [2..n] do
3      for j ∈ [p1..P] do
4          if j < pi then
5              U(i, j) ← U(i − 1, j)
6          else
7              if j = pi then
8                  U(i, j) = max{ui, U(i − 1, j)}
9              else
10                 U(i, j) = max{U(i − 1, j), U(i − 1, j − pi) + ui}
```

La meilleure utilité est alors U(n, P)

2.2.6 Comment récupérer une solution optimale ?

Il faut pour cela stocker dans chaque case (i, j) (en plus de U(i, j)) la façon dont est obtenu le minimum, c'est-à-dire :

soit $\mathbf{x_i = 0}$ lorsque U(i, j) = U(i-1, j)

soit $\mathbf{x_i = 1}$ lorsque U(i, j) = U(i -1; j-p_i) + u_i

Ainsi en « remontant » dans le tableau ligne après ligne à partir de la dernière case, U(n - P), on a un vecteur optimal (x_1, x_2,…,x_n), ou politique optimale

Cela nécessite n étapes supplémentaires

L'utilité maximale est obtenue avec les deux premiers objets

Estimations des complexités

En temps

si on trie les poids il y a O(n log n) étapes préliminaires

Chaque ligne nécessite P p_1 + 1 étapes (pour chaque case du tableau on calcule un minimum entre deux valeurs, le coût est donc constant) et il y a n lignes Globalement, la complexité en temps est donc en O(n P), donc si P est « petit » l'algorithme peut être efficace

En espace

le tableau a au plus nP cases, chacune contenant deux valeurs (U(i,j) et le x_i réalisant le minimum) la complexité en espace est donc aussi en O(n P)

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

3. RESULTATS ET DISCUSSION

3.1 Contribution : Adaptation de l'algorithme basé sur le problème de Sac à dos traité par la programmation dynamique pour l'Analyse de la valeur à optimiser

Ce genre de problème a un point commun avec le problème de sac à doc rencontré en recherche opérationnelle notamment dans le problème associé à la programmation linéaire mais la différence réside sur l'organisation. Dans le cas présent l'obtention de la valeur optimale minimale est essentielle tandis que l'autre est la valeur optimale maximale. Obtention de la valeur optimale minimale des coûts, tandis que pour les fonctions une valeur optimale maximale

Ensuite, ici chaque constituant est à optimiser par exemple si une sous-politique possède n possibilités on doit choisir une qui est le résultat de l'algorithme, considérée comme optimale. De plus si le système dans son ensemble possède m sous politiques on doit trouver ces m valeurs optimales. Le résultat final n'est que la combinaison de ces m valeurs obtenues en fonction de la gestion des ordonnancements des tâches associées.

On dresse alors l'ensemble de sous-structure puis chaque sous-structure retenue est optimale

3.1.1 Cas de Sous-Structure optimale.

Supposons que l'on ait une solution optimale I_{MAX}. On peut distinguer deux cas :

– soit $n \in I_{MAX}$ et dans ce cas, I_{MAX} (C)\{n} est une solution optimale pour le problème où le sac à dos a pour contenance maximale $C - p_N$ et où l'on cherche à le remplir avec les n – 1 premiers objets (de poids respectifs $p_1, ..., p_{N-1}$ et de valeurs $v_1, ..., v_{N-1}$). En effet, si on avait une solution meilleure I' pour ce dernier problème, le sous-ensemble d'indices $I' \cup \{n\}$ serait une solution qui contredirait l'optimalité de la solution I_{MAX}.

– soit $n \notin I_{MAX}$ et dans ce cas, I_{MAX} (C) est une solution optimale pour le problème où la Valeur a pour coût maximale C et où l'on cherche à le remplir avec les n − 1 premiers objets.

Car comme précédemment, si on avait une solution meilleure I' pour ce dernier problème, le sous-ensemble d'indices I' serait une solution qui contredirait l'optimalité de la solution I_{MAX}.

3.1.2 Cas de traitement de la relation de récurrence pour trouver la valeur optimale que l'on peut cumuler.

Soit $p = (p_1, ..., p_N)$ et $v = (v_1, ..., v_N)$ deux séquences de n nombres entiers positifs. Nous notons pour tous $i \leq n$ et $C \geq 0$, $P_I(C)$ le problème de l'AV dont les données initiales sont C pour le coût maximal de la somme pour la fonction objective et $(p_1, ..., p_I)$ et $(v_1, ..., v_I)$ pour les deux suites respectivement des points de fonctions et des valeurs des i objets du problème. De plus, on note $I_I(C)$ un sous-ensemble d'indices qui permet d'obtenir la solution optimale du problème $P_I(C)$ et on note $M_I(C)$ la valeur minimale transportable $M_I(C) = v_I$.

On regarde maintenant la structure de $I_I(C)$: soit $i \in I_I(C)$ et dans ce cas, $I \in I_I$ (C) $I_I(C)\{i\}$ est une solution optimale pour le problème $P_{I-1}(C - p_I)$, soit $i \in / I_I(C)$ et dans ce cas $I_I(C)\{i\}$ est une solution optimale pour $P_{I-1}(C)$. Comme on ne sait pas à l'avance si i appartient ou non à $I_I(C)$, on est obligé de regarder quelle est la meilleure des deux solutions. On a donc que :

$M_I(C) = \min\{M_{I-1}(C), M_{I-1}(C - p_I) + v_I\}$

3.2 Algorithme pour la valeur optimale.

On va commencer par calculer le coût optimal par une approche itérative. L'entrée est deux tableaux P [1..n] et V [1..n] contenant respectivement les valeurs $p_1, ..., p_N$ et $v_1, ..., v_N$ et une valeur B représentant la somme maximale à ne pas dépasser. La procédure utilise un tableau auxiliaire M [0..n, 0..B] pour mémoriser les coûts $M_I(C)$ et un tableau X[1..n, 1..B] dont l'entrée (i, C) contient un booléen qui indique si i appartient ou non à une solution optimale du problème $P_I(C)$.

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

Algorithme 1 : VALEURMIN

Entrées : Deux tableaux d'entiers positifs P [1..n] et V [1..n] et B un nombre entier positif.

Sorties : Rien

```
1.  VALEURMIN(P, V, B)
2.  n := longueur[P ] ;
3.  pour C allant de 1 à  B faire
4.  M[0, C] := 0
5.  pour i allant de 1 à  n faire
6.  M [i, 0] := 0
7.  pour C allant de 1 à  B faire
8.  pour i allant de 1 à  n faire
9.  q1 := M [i − 1, C];
10. si C − P [i] < 0 alors
11. q2 := −∞
12. sinon
13. q2 := M [i − 1, C − P [i]] + V [i]
14. si q1 < q2 alors
15. M [i, C] := q2; X[i, C] := vrai
16. sinon
17. M [i, C] := q1 ; X[i, C] := faux
18. Fin
```

Analyse de l'algorithme :

Preuve de Terminaison

L'algorithme converge car on est en présence de boucles finies

Preuve de Validité :

L'algorithme commence aux lignes 3-6 par l'initialisation des cas de base. S'il n'y a pas d'objet à prendre (i = 0), la valeur maximale transportable est évidement 0 quel que soit le coût de la fonction. Donc M [0, C] := 0, pour C = 0, 1, ..., B. De même si la fonction a une valeur nulle, on ne peut prendre aucun objet. Donc M [i, 0] := 0, pour i = 1, ..., n. On utilise ensuite lignes 7-17 la récurrence pour calculer M [i, C] pour i = 1, 2, ..., n et C = 1, ..., B. On remarque que, quand l'algorithme en est au calcul de M [i, C], il a déjà calculé les valeurs M [i − 1, C] et M [i − 1, C − p_I] si i > 0 et C − $p_I \geq 0$. Si C − $p_I < 0$, cela veut dire que la fonction de point p_I est trop lourde pour être rajouté dans le sac. Dans ce cas, on accepte alors la prise de cet objet (M [i, C] = M [i − 1, C] car $q_1 > q_2$ dans tous les cas). Quand on accède à la ligne 15, on a M [i − 1, C − p_I] > M [i − 1, C]. Ceci veut dire que l'on peut prendre la i-^{ième} fonction dans la liste pour construire une solution optimale.

3.2.1 *Analyse de la Complexité* en nombre de comparaisons :

Le nombre de comparaisons est nB (une comparaison à chaque passage dans la double boucle ligne 7-8). La procédure VALEURMIN fait donc Θ(nB) comparaisons. L'algorithme nécessite un espace de stockage Θ(nB) pour le tableau M . VALEURMIN est donc beaucoup plus efficace que la méthode en temps exponentiel consistant à énumérer tous les sous-ensembles d'indice possibles et à tester chacun d'eux. Néanmoins l'algorithme du sac à dos est ce que l'on appelle un algorithme pseudo-polynomial. En fait, la valeur B peut être stockée en machine sur $\lceil \log_2 B \rceil$ bits et donc le nombre de comparaisons nB est exponentiel par rapport à la taille mémoire pour stocker B. On peut montrer de plus que le problème de l'AV est un problème NP-complet.

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

3.2.2 Construction d'une solution optimale.

Bien que VALEURMIN détermine la valeur minimale que peut être acceptée, elle ne rend pas le sous-ensemble d'indices qui permet d'obtenir la valeur minimale. Le tableau X construit par VALEURMIN peut servir à retrouver rapidement une solution optimale pour le problème P_N (B). On commence tout simplement en X[n, B] et on se déplace dans le tableau grâce aux booléens de X. Chaque fois que nous rencontrons X[i, C] = faux, on sait que i n'appartient pas à une solution optimale. Dans ce cas, on sait qu'une solution optimale pour le problème P_{I-1}(C) est une solution optimale pour le problème P_I(C). Si l'on rencontre X[i, C] = vrai, alors une solution optimale pour le problème P_{I-1}(C − P [i]) peut être étendue à une solution optimale p our le problème P_I(C) en lui rajoutant i. Ceci permet d'écrire le pseudo-code récursif suivant :

Algorithme 2 : AFFICHER-INDICE

Entrées : les tableaux X et P et deux entiers i, C

Sorties : Rien mais affiche une solution optimale

```
Debut
1       AFFICHER-INDICE(X, P, i, C)
2       si i = 0 alors
3           Stop
4       si X[i, j] = faux alors
5           AFFICHER-INDICE(X, P, i − 1, C);
6       sinon
7           AFFICHER-INDICE(X, P, i − 1, C − P [i]);
8           afficher(i);
9 Fin
```

Nous proposons maintenant une version récursive de l'algorithme. Elle est composée de deux parties, une partie initialisation (MEMORISATION-VALEURMIN) et une partie construction Récursive (RECUPERER-VALEURMIN).

Algorithme 3 : MEMORISATION-VALEURMIN

Entrées : Deux tableaux d'entiers positifs P [1..n] et V [1..n] et B un nombre entier positif.

Sorties : La valeur optimale que l'on peut transporter.

```
1  MEMORISATION-VALEURMIN(P, V, B)
2       n := longueur[P ]
3       Créer un tableau d'entiers M [0..n, 0..B];
4       Créer un tableau de booléens X[1..n, 1..B];
5       pour i allant de 1 à  n faire
6           pour C allant de 1 à  B faire
```

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

```
7            |        M [i, C] := −1 ; X[i, C] := faux
8        retourner RECUPERER-VALEURMIN (P, V, n, B)
9   Fin
```

Analyse de l'algorithme :

Preuve de Terminaison :

Il suffit de remarquer que l'on travaille avec une première boucle de n répétitions et d'une deuxième boucle B répétition c'est-à-dire nB comparaisons et en plus RECUPERER-VALEURMIN est déjà convergent.

Preuve de Validité :

Immédiat par définition directe de M [i, C].

Analyse de la Complexité en nombre de comparaisons :

Il est facile de montrer que l'on fait nB comparaisons pour pouvoir récupérer la valeur minimale.

Algorithme 4 : RECUPERER-VALEURMIN

Entrées : Deux tableaux d'entiers positifs P [1.n] et V [1.n] et deux nombres entiers positifs i et C.

Sorties : Un entier indiquant la valeur maximale que peut transporter le sac de contenance C pour des objets pris parmi les i premiers

```
1       RECUPERER-VALEURMIN(P, V, i, C)
2       si M [i, C] ≥ 0 alors
3               retourner M [i, C]
4       sinon
5               si i = 0 ou C = 0 alors
6                       M [i, j] := 0
7               sinon
8                       q1 := RECUPERER-VALEURMIN(P, V, i − 1, C);
 9        si C − P [i] < 0 alors
10           q2 := −∞
11        sinon
12          q2 :=RECUPERER-VALEURMIN(P, V, i − 1, C − P [i]) + V [i]
13        si q1 < q2 alors
14           M [i, C] := q2; X[i, C] := vrai
15        sinon
16           M [i, C] := q1 ; X[i, C] := faux
17  retourner M [i, C]
```

International Journal of Progressive Sciences and Technologies (IJPSAT)
ISSN: 2509-0119.

https://ijpsat.org/
Vol. 53 No. 1 October 2025, pp. 476-488

Analyse de l'algorithme :

Preuve de Terminaison :

Il suffit de remarquer que l'on appelle récursivement RECUPERER-VALEURMIN sur des instances où i a diminué de 1 et que l'algorithme se termine quand i = 0.

Preuve de Validité :

Immédiat par définition récursive de M [i, C].

Analyse de la Complexité en nombre de comparaisons :

Il est facile de montrer que l'on fait moins de nB comparaisons car on remplit au plus une fois chaque case du tableau M. Mais dans cette version récursive le tableau M n'est pas nécessairement entièrement calculé. Pour terminer, un ensemble de procédure est à mettre place pour traiter la recherche de chaque sous politique optimale puis de réaliser leur réunion pour donner la solution de la politique voulue.

Dans ce cas, on a besoin le résultat de l'analyse fonctionnelle avec les possibilités de familles de solutions proposées pour chaque groupe de fonctions. D'où la spécification suivante : Soit F un tableau d'éléments de l'analyse dont la taille est égale au nombre de résultat obtenu pendant l'analyse fonctionnelle. Ensuite chaque fonctionnalité numéro I obtenu possède n solutions possibles dont on calculera la valeur minimale, et la solution de I pourra être un ensemble d'élément ou un unique élément selon le résultat de la recherche minimale. Et le résultat de l'ensemble est la somme de toutes les valeurs des I considérées, d'où l'algorithme AV_TROUVE

Analyse de l'algorithme :

Algorithme 5 AV_TROUVE

Entrées : les nombres de fonctions à vérifier NB_f

Le seuil maximal à respecter V_s

Le nombre d'éléments pour chaque groupe Elt[1..NB_f]

Les valeurs proposées par chaque fonction dans un groupe Prop[1..Elt[i]] pour i variant de 1 à NB_f.

Sorties : Les fonctions retenues avec les valeurs associées ainsi que le total Final de la valeur.

```
1     AV_TROUVE(NB_f,V_s,Prop)
2     Final=0
3     F={}
4     Pour j  allant de 1 jusqu'à NB_f faire
5         Pour k  allant de 1 jusqu'à Elt[j]  faire
6             V[j]=RECUPERER_VALEURMIN(NB_f,Elt[k],Prop[k],V_s)
7             F=FU{V[j]}
8             Final =Final +Prop[k];
9         Fin pour
10    Fin pour
11    retourner (F ,V)
```

Preuve de Terminaison :

Il suffit de remarquer que l'on appelle récursivement AV_TROUVE pour RECUPERER_VALEURMIN NB_f x NB_f fini par intermédiaire des procédures internes qui à leurs tours sont toutes convergentes.

Preuve de Validité :

L'expression utilisée justifie que le résultat va être obtenu par la création successive de la solution sous ensemble.

4. CONCLUSION

L'adaptation de l'algorithme du problème de Sac à Dos, basée sur la programmation dynamique en intervertissant le rôle pour calculer l'optimisation maximale par(ou) l'optimisation minimale apporte une nouvelle activité sur la façon et les moyen de travailler en Analyse de la Valeur même si la démarche associée a déjà des normes, alors que le choix final des fonctions choisies demande des apports ou des efforts de la personne qui gère la situation c'est pourquoi nous avons apporté notre contribution pour résoudre le problème. Afin de vérifier la qualité de la contribution, des études comparatives avec d'autres algorithmes comme diviser pour régner, algorithme glouton feront l'objet des études à l'avenir.

REFERENCES

[1] Thomas H. C, Charles E., Ronald L. R., Clifford S., Introduction to Algorithms, Second Edition, The MIT Pres, Cambridge, Massachusetts London, England,2001,984 pages

[2] Thècle A, Gregory Z., Nicolas P. Modèle d'analyse de la valeur de l'innovation des systèmes de produit – service, UNIVERSITE DE BORDEAUX, 2017,8 pages

[3] Yoch, Introduction à la programmation dynamique, openclassroom,2012, 12 pages

[4] Laurent Lemarchand, Algorithmique avancée et parallélisme, LISyC/UBO, Université de Brest, 2015, 62 pages

[5] Gauthier Picard, Paradigmes algorithmiques Quelques méthodes de conception d'algorithmes, Mines Saint Etienne,2012,115 pages

[6] Sameh Grainia, L'algorithme de Branch and Price and Cut pour le problème de conception de réseaux avec coûts fixes et sans capacité, Département d'informatique et de recherche opérationnelle Faculté des arts et des sciences, 2015, 91 pages

[7] Hidri Dounia, Adaptation de la méthode du branch and bound au problème d'affectation, Université Abdelmalek,2015, 14 pages