COURS : PROGRAMMATION DYNAMIQUE = ALGORITHME DE BELLMAN-FORD=

Notre sixième étude a pour but d'introduire un algorithme de calcul du plus court chemin dans un graphe à l'aide de la programmation dynamique. L'algorithme de Bellman-Ford résout le problème du plus court chemin à partir d'une source unique en présence d'arêtes de longueur négative ; il présente également l'avantage d'être « plus distribué » que l'algorithme de Dijkstra et, pour cette raison, a profondément influencé la manière dont le trafic est acheminé sur Internet.

I) PLUS COURTS CHEMINS AVEC DES LONGUEURS D'ARÊTES NÉGATIVES

I.1. Le plus court chemin à partir d'une source unique

Dans le problème du plus court chemin à partir d'une source unique, l'entrée est constituée d'un graphe orienté G = (V, E) avec une longueur réelle ℓ_e pour chaque arête e ∈ E, et d'une origine désignée s ∈ V, appelée le sommet source ou sommet de départ. La longueur d'un chemin est la somme des longueurs de ses arêtes.

Le rôle d'un algorithme est de calculer, pour chaque destination possible v, la longueur minimale dist(s, v) d'un chemin orienté dans G allant de s à v (si un tel chemin n'existe pas, dist(s, v) est définie comme valant + ∞). Par exemple, les distances de plus court chemin depuis s dans le graphe figure 1 sont dist(s, s) = 0, dist(s, v) = 1, dist(s, w) = 3, et dist(s, t) = 6 :

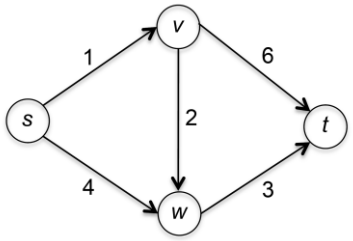


Figure 1 : Exemple de graphe orienté

Par exemple, si chaque arête e a une longueur unitaire $\ell_e = 1$, un plus court chemin minimise le nombre de sauts (c'est-à-dire le nombre d'arêtes) entre son origine et sa destination. Si le graphe représente un réseau routier et que la longueur de chaque arête est le temps de trajet attendu d'une extrémité à l'autre, alors le problème du plus court chemin à partir d'une source unique consiste à calculer les temps de conduite depuis une origine (le sommet source) vers toutes les destinations possibles.

Vous connaissez déjà l'algorithme de Dijkstra pour le cas particulier du problème du plus court chemin à partir d'une source unique dans lequel chaque longueur d'arête est non négative. Mais l'algorithme de Dijkstra n'est pas toujours correct dans les graphes comportant des longueurs d'arêtes négatives. Il échoue même dans un exemple aussi trivial que celui de la figure 2 (ici, l'algorithme de Dijkstra donnerait une valeur minimale de -2 au lieu de -4) :

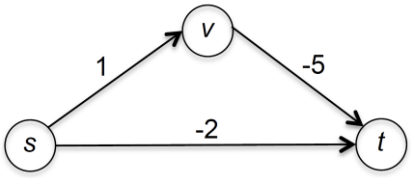


Figure 2 : Exemple d'un graphe orienté avec des arêtes de valeurs négatives

Si nous voulons prendre en charge des longueurs d'arêtes négatives, il nous faudra un nouvel algorithme de plus court chemin.

Remarque : on ne peut pas réduire le problème du plus court chemin à partir d'une source unique avec des longueurs d'arêtes quelconques au cas particulier des longueurs d'arêtes non négatives en ajoutant une grande constante positive à la longueur de chaque arête. Dans l'exemple à trois sommets de la figure 2, ajouter 5 à la longueur de chaque arête ferait passer le plus court chemin de s → v → t à s → t.

I.2. Le problème des cycles négatifs

Dans de nombreuses applications, comme le calcul d'itinéraires routiers, les longueurs d'arêtes sont automatiquement non négatives et il n'y a donc pas lieu de s'inquiéter. Mais les chemins dans un graphe peuvent représenter des séquences abstraites de décisions. Par exemple, on peut vouloir calculer une séquence rentable de transactions financières impliquant à la fois des achats et des ventes. Ce problème correspond à la recherche d'un plus court chemin dans un graphe dont les longueurs d'arêtes sont à la fois positives et négatives.

Avec des longueurs d'arêtes négatives, il faut faire attention à ce que l'on entend même par « distance de plus court chemin ». C'est suffisamment clair dans l'exemple précédent de la figure 2 : dist(s, s) = 0, dist(s, v) = 1 et dist(s, t) = −4. Mais cela se complique dans un graphe comme celui de la figure 3 :

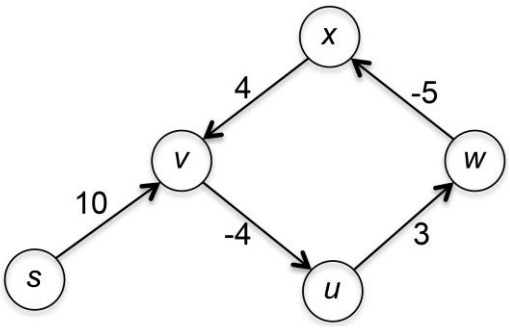


Figure 3 : Exemple de graphe orienté avec un cycle négatif

Le problème est que ce graphe contient un cycle négatif, c'est-à-dire un cycle orienté pour lequel la somme des longueurs des arêtes est négative. Que pourrait-on entendre par « plus court chemin de s à v » ?

Option n°1 : autoriser les cycles.

La première option consiste à autoriser des chemins de s à v qui incluent un ou plusieurs cycles. Mais alors, en présence d'un cycle négatif, un « plus court chemin » peut même ne pas exister. Par exemple, dans le graphe ci-dessus, il existe un chemin de s à v en un seul saut, de longueur 10. En ajoutant à la fin une traversée de cycle, on obtient un chemin de s à v en cinq sauts, de longueur totale 8. En ajoutant une deuxième traversée, on augmente le nombre de sauts à 9 mais on diminue la longueur totale à 6… et ainsi de suite, à l'infini.

Ainsi, il n'existe pas de plus court chemin de s à v, et la seule définition raisonnable de dist(s, v) est -∞.

Option n°2 : interdire les cycles.

Sans répétition de sommets autorisée, il n'y a qu'un nombre fini de chemins à prendre en compte. Le « plus court chemin de s à v » serait alors celui dont la longueur est la plus petite. Cette définition est parfaitement cohérente, mais il existe un problème plus subtil : en présence d'un cycle négatif, cette version du problème du plus court chemin à partir d'une source unique est ce qu'on appelle un « problème NP-difficile ». Ce sont des problèmes pour lesquels on ne connaît pas d'algorithme qui soit garanti correct et qui s'exécute en temps polynomial.

En interdisant les cycles, on impose la contrainte globale « ne pas repasser par un sommet » qui rapproche le problème de problèmes combinatoires difficiles. Avec des poids négatifs, cette contrainte change totalement la nature du problème : on ne peut plus s'appuyer sur les propriétés qui rendent Dijkstra/Bellman-Ford efficaces.

Nous allons donc résoudre le problème du plus court chemin à partir d'une source unique pour des graphes qui ne contiennent aucun cycle négatif, comme l'exemple à trois sommets de la figure 2.

II) DÉFINITION DU PROBLÈME

L'algorithme de Bellman-Ford résout le problème du plus court chemin à partir d'une source unique dans des graphes comportant des longueurs d'arêtes négatives, en ce sens qu'il calcule soit les distances de plus court chemin correctes, soit détermine correctement que le graphe d'entrée contient un cycle négatif.

Cet algorithme découle naturellement du schéma de conception que nous avons utilisé dans nos autres études de cas en programmation dynamique.

Problème du plus court chemin à partir d'une source unique

Entrée : Un graphe orienté $G = (V, E)$, un sommet source $s \in V$, et une longueur réelle ℓ_e pour chaque arête $e \in E$.

Sortie : l'un des résultats suivants :

- la distance de plus court chemin $dist(s, v)$ pour chaque sommet $v \in V$; ou
- une déclaration indiquant que G contient un cycle négatif.

Bellman-Ford est surtout utilisé dès qu'on a des poids négatifs possibles et/ou le besoin explicite de détecter un cycle négatif. Voici quelques exemples :

- Routage dans les réseaux : chaque routeur maintient une estimation de la « distance » (coût) vers chaque destination et l'améliore à partir des informations de ses voisins.
- Détection d'arbitrage en finance : Pour modéliser des échanges de devises, on construit un graphe où les arêtes représentent des taux de change, puis on transforme les poids. Une opportunité d'arbitrage correspond alors à un cycle de poids total négatif. Bellman-Ford est utile parce qu'il détecte précisément ce type de cycles.
- Dans certains modèles de planification temporelle, utiliser Bellman-Ford sur le graphe des contraintes sert à détecter des incohérences (cycles négatifs).

Pour résoudre ce problème de manière exhaustive (par brute force) et trouver la plus courte distance d'une source s vers une destination v, il faudrait énumérer tous les chemins simples de s vers chaque v, calculer la longueur de chacun (somme des arêtes) et prendre le minimum.

Dans un graphe orienté très dense, on peut permuter au plus les (n-2) sommets intermédiaires, et le nombre de chemins simples de s à un sommet v est majoré par $C \cdot (n-2)!$

(où C est une constante et n le nombre de sommets). Si on recalcule la somme des poids en parcourant le chemin (coût en O(n) par chemin), on obtient une complexité totale en O(n·(n-2)!) = O(n!). C'est un problème à complexité factorielle, donc beaucoup plus grande qu'exponentielle, qui demande une approche plus efficace.

Si en plus on souhaite obtenir les distances les plus courtes depuis la source s vers chacun des sommets qui composent le graphe, la complexité augmente en $O(n^2 \cdot (n-2)!) = O(n \cdot n!)$.

Remarque : si on autorise les cycles et qu'on fait tous les chemins sans borne, ce n'est même pas un algorithme fini : il y a une infinité de chemins possibles (on peut boucler arbitrairement), donc un algorithme exhaustif ne termine pas en général.

III) SOUS-STRUCTURE OPTIMALE ET RELATION DE RÉCURRENCE

III.1. Recherche des sous-problèmes

Comme toujours en programmation dynamique, l'étape la plus importante consiste à comprendre les différentes manières dont une solution optimale peut être construite à partir de solutions optimales à des sous-problèmes plus petits.

Une première intuition pourrait être que les sous-problèmes doivent correspondre à des sous-graphes du graphe d'entrée initial, avec une taille de sous-problème égale au nombre de sommets ou d'arêtes dans le sous-graphe. Cette idée a bien fonctionné pour le problème WIS sur des graphes en chemin (voir le cours d'introduction à la programmation dynamique), où les sommets étaient intrinsèquement ordonnés et où il était relativement clair sur quels sous-graphes se concentrer. Cependant, avec un graphe général, en revanche, il n'existe pas d'ordre intrinsèque des sommets ou des arêtes.

L'algorithme de Bellman-Ford adopte une approche différente. Intuitivement, on peut s'attendre à ce qu'un préfixe P' d'un plus court chemin P soit lui-même un plus court chemin, mais vers une destination différente :

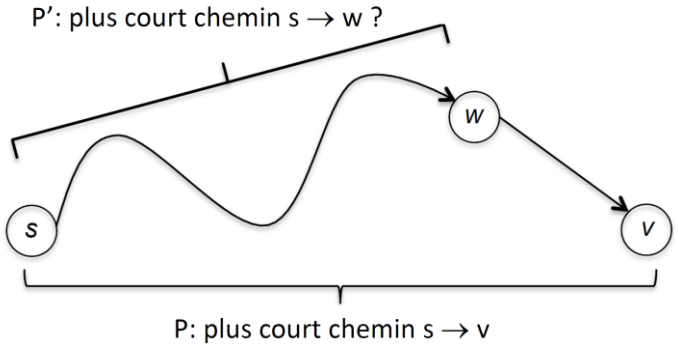


Figure 4 : Le préfixe P' du plus court chemin P (s→v) est-il lui-même un plus court chemin vers w ?

Cependant, avec des longueurs d'arêtes négatives, la longueur de P' pourrait être supérieure à celle de P. En revanche, P' contient moins d'arêtes que P, ce qui motive l'idée ingénieuse à la base de l'algorithme de Bellman-Ford : introduire un paramètre i de nombre de sauts qui restreint artificiellement le nombre d'arêtes autorisées dans un chemin, les sous-problèmes « plus grands » disposant d'un budget d'arêtes i plus élevé. Ainsi, un préfixe de chemin peut effectivement être considéré comme la solution d'un sous-problème plus petit.

Prenons par exemple le graphe de la figure 5 ci-dessous et pour la destination v, les sous-problèmes correspondant à des valeurs successives du budget d'arêtes i. Définissons $L_{i,v}$ comme la longueur d'un plus court chemin de s à v comportant au plus i arêtes.

- Lorsque i vaut 0 ou 1, il n'existe aucun chemin de s à v comportant au plus i arêtes, et il n'y a donc pas de solution aux sous-problèmes correspondants. La distance de plus court chemin sous la contrainte du nombre de sauts est alors $L_{0,v} = L_{1,v} = +\infty$.
- Lorsque i = 2, il existe un unique chemin de s à v comportant au plus i arêtes ($s \rightarrow t \rightarrow v$), et la valeur du sous-problème est $L_{2,v} = 4$.
- Lorsque i = 3 (ou davantage), le chemin $s \rightarrow u \rightarrow w \rightarrow v$ devient admissible et fait baisser la distance de plus court chemin de 4 à 3 : $L_{3,v} = 3$

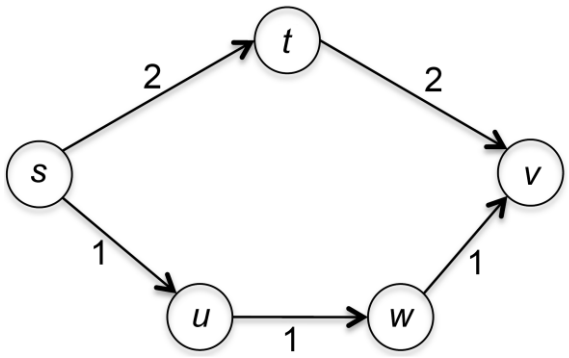


Figure 5 : Exemple de graphe

Les chemins comportant des cycles sont autorisés comme solutions d'un sous-problème. Si un chemin utilise une arête plusieurs fois, chaque utilisation est comptabilisée dans son budget de nombre de sauts. Une solution optimale pourrait très bien parcourir un cycle négatif encore et encore, mais elle finira par épuiser son budget d'arêtes (fini). Pour une destination v fixée, l'ensemble des chemins admissibles s'agrandit avec i, et ainsi $L_{i,v}$ ne peut que diminuer lorsque i augmente.

Contrairement à nos études de cas précédentes en programmation dynamique, chaque sous-problème travaille avec l'entrée complète (plutôt qu'avec un préfixe ou un sous-ensemble de celui-ci) ; le caractère ingénieux de ces sous-problèmes réside dans la manière dont ils contrôlent la taille autorisée de la sortie.

Tel qu'énoncé, le paramètre i pourrait être un entier positif arbitrairement grand, et il existerait donc une infinité de sous-problèmes. Cependant, nous verrons bientôt qu'il n'y a aucune raison de s'occuper des sous-problèmes pour lesquels i est supérieur à n, le nombre de sommets, ce qui implique qu'il y a $O(n^2)$ sous-problèmes pertinents.

Sous-problèmes de l'algorithme de Bellman-Ford

Calculez $L_{i,v}$, la longueur d'un plus court chemin de s à v dans G comportant au plus i arêtes, les cycles étant autorisés. (Si un tel chemin n'existe pas, définir $L_{i,v} = +\infty$)

(Pour chaque i = 0, 1, 2 … et chaque $v \in V$)

III.2. Sous-structure optimale

Considérons un graphe d'entrée G = (V, E) avec un sommet source s ∈ V, et fixons un sous-problème, défini par un sommet destination v ∈ V et une contrainte de nombre de sauts i ∈ {1, 2, 3, …}.

Supposons que P soit un chemin de s à v comportant au plus i arêtes, et qu'il soit de plus le plus court parmi ces chemins. À quoi doit-il ressembler ? Deux cas sont possibles :

Cas n°1 : P comporte (i - 1) arêtes ou moins.
Dans ce cas, le chemin P peut immédiatement être interprété comme une solution au sous-problème plus petit ayant pour budget d'arêtes (i – 1) (avec toujours pour destination v).

Cas n°2 : P comporte i arêtes.
Soit L la longueur de P. Soit P' les (i - 1) premières arêtes de P, et (w, v) son dernier saut de longueur ℓ_{wv}. Le préfixe P' est alors un chemin optimum de s à w comportant au plus (i - 1) arêtes et de longueur L - ℓ_{wv} (sinon on pourrait améliorer P en remplaçant le préfixe P' par un meilleur, et obtenir un chemin s → v encore plus court, ce qui est contradictoire).

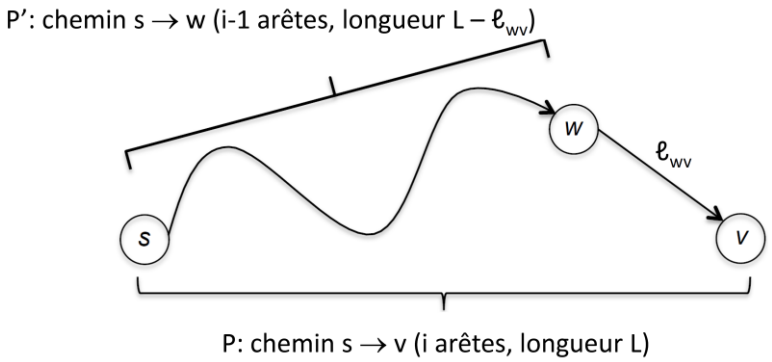


Figure 6 : Illustration du cas n°2

Exemple : Soit le graphe ci-dessous (figure 7). On fixe i = 3 (au plus 3 arêtes) :

Cas n°1 : Le chemin P1 : s → a → v a 2 arêtes et sa longueur vaut 3. Pour le sous problème « au plus 3 arêtes », P1 est une solution optimale. Comme P1 utilise 2 arêtes et qu'il est solution optimale au problème « au plus 3 arêtes », il est aussi une solution optimale au problème « au plus 2 arêtes ».

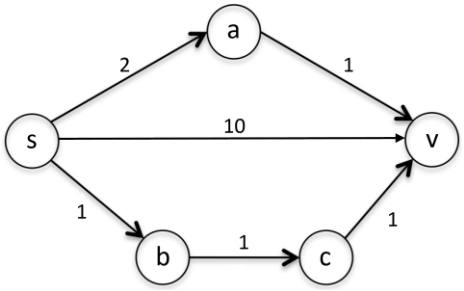


Figure 7 : Exemple de graphe

Cas n°2 : Le chemin P2 : s → b → c → v a exactement 3 arêtes et sa longueur vaut 3. Si on retire la dernière arête c → v, on obtient le préfixe P'2 : s → b → c, le chemin de s vers c, avec au plus (i - 1) = 2 arêtes, de longueur 3 - ℓ_{cv} = 3 – 1 = 2.

Ce que formalise Bellman-Ford est que si un plus court chemin P : s $\rightarrow$ v avec budget i utilise exactement i arêtes et se termine par (w $\rightarrow$ v), alors son préfixe P' : s $\rightarrow$ w doit être un plus court chemin vers w avec budget (i - 1).

III.3. Équation de récurrence sur les valeurs optimales

Comme d'habitude, l'étape suivante consiste à synthétiser notre compréhension de la sous-structure optimale en une récurrence qui met en œuvre une recherche exhaustive parmi les candidats possibles à une solution optimale.

On note $L_{i,v}$ la longueur minimale d'un chemin de s à v comportant au plus i arêtes, les cycles étant autorisés (s'il n'existe pas de tels chemins, alors $L_{i,v} = +\infty$).

Récurrence sur la valeur de la solution optimale

Les cas de base sont (s est le sommet de départ) :

- $L_{0,s} = 0$ (il existe un chemin de s $\rightarrow$ s utilisant 0 arêtes – chemin vide de longueur 0)
- $L_{0,v} = +\infty$ pour tout $v \neq s \in V$ (avec 0 arêtes, on ne peut atteindre aucun autre sommet autre que s)

Pour tout $i \geq 1$ et tout $v \in V$:

$$L_{i,v} = min \begin{cases} L_{i-1,v} & (cas\ n°1) \\ min_{(w,v) \in E} \{ L_{i-1,w} + \ell_{w,v} \} & (cas\ n°\ 2) \end{cases}$$

Le « min » externe dans la récurrence met en œuvre la recherche exhaustive entre le cas 1 et le cas 2. Le « min » interne met en œuvre la recherche exhaustive, à l'intérieur du cas 2, sur tous les choix possibles pour le dernier saut d'un plus court chemin.

Si $L_{i-1,v}$ et tous les $L_{i-1,w}$ pertinents valent $+\infty$, alors v est inatteignable depuis s en i sauts ou moins, et on interprète la récurrence comme $L_{i,v} = +\infty$.

III.4. Critère d'arrêt

Dans les sous-problèmes que nous avons définis, le budget d'arêtes i peut être un entier positif arbitrairement grand, ce qui signifie qu'il existe une infinité de sous-problèmes. Comment savoir quand s'arrêter ?

Un bon critère d'arrêt découle de l'observation suivante : les solutions d'un lot donné de sous-problèmes, pour un budget d'arêtes fixé i et un sommet v, ne dépendent que des solutions du lot précédent (budget i - 1). Ainsi, si un lot de sous-problèmes a à un moment exactement les mêmes solutions optimales que le lot précédent (le cas 1 de la récurrence l'emportant pour chaque destination), alors ces solutions optimales resteront identiques pour toujours.

On peut ainsi définir le critère d'arrêt de Bellman-Ford :

Critère d'arrêt de Bellman-Ford

Si pour un certain k on a, pour tout v ∈ V :

$$L_{k+1,v} = L_{k,v}$$

… alors :

- $L_{i,v} = L_{k,v}$ pour tout i ≥ k et toute destination v ; et
- Pour toute destination v, $L_{k,v}$ est la distance de plus court chemin correcte dist(s, v) de s → v dans G.

Ce critère d'arrêt promet qu'il est sans danger de s'arrêter dès que les solutions des sous-problèmes se stabilisent, c'est-à-dire dès que $L_{k+1,v} = L_{k,v}$ pour un certain k ≥ 0 et pour tout v ∈ V.

Mais cela finira-t-il par se produire ? En général, non. Toutefois, si le graphe d'entrée ne contient aucun cycle négatif, les solutions des sous-problèmes sont garanties de se stabiliser au plus tard lorsque i atteint n, le nombre de sommets.

On peut ainsi définir le critère d'arrêt de Bellman-Ford pour des graphes sans cycle négatif :

Critère d'arrêt de Bellman-Ford pour des graphes sans cycle négatif

En supposant que le graphe d'entrée G ne contient aucun cycle négatif, on a :

$$L_{n,v} = L_{n-1,v}$$

pour toute destination v, où n est le nombre de sommets du graphe d'entrée.

Cela montre que si le graphe d'entrée ne contient pas de cycle négatif, alors les solutions des sous-problèmes se stabilisent au plus tard au n-ième lot. Ou, sous forme contraposée : si les solutions des sous-problèmes ne se stabilisent pas au plus tard au n-ième lot, alors le graphe d'entrée contient bien un cycle négatif.

Ensemble, ces deux derniers critères d'arrêt indiquent quel est le dernier lot de sous-problèmes dont il vaut la peine de s'occuper : celui correspondant à i = n.

Si les solutions des sous-problèmes se stabilisent (avec $L_{n,v} = L_{n-1,v}$ pour tout v ∈ V), alors le premier critère implique que les valeurs $L_{n-1,v}$ sont les distances de plus court chemin correctes.

Si les solutions des sous-problèmes ne se stabilisent pas (avec $L_{n,v} = L_{n-1,v}$ pour au moins un sommet v ∈ V), alors la contraposée du deuxième critère implique que le graphe d'entrée G contient un cycle négatif, auquel cas l'algorithme est dispensé de calculer des distances de plus court chemin et retourne qu'un cycle négatif a été détecté.

Le plus grand sous-problème utile qu'il est nécessaire de calculer pour obtenir la distance optimale de s vers v est donc (n-1, v). Le sous-problème (n,·) sert à la détection de cycle négatif.

IV) SOUS-PROBLÈMES ET COMPLEXITÉ

IV.1. Définition des sous-problèmes

Rappelons ici les sous-problèmes en prenant en compte les critères d'arrêts :

Sous-problèmes de l'algorithme de Bellman-Ford

Calculez $L_{i,v}$, la longueur d'un plus court chemin de s à v dans G comportant au plus i arêtes, les cycles étant autorisés. (Si un tel chemin n'existe pas, définir $L_{i,v} = +\infty$)

(Pour chaque i = 0, 1, 2 … n et chaque $v \in V$)

Le plus grand sous-problème utile pour les distances est (n-1, v). La ligne (n,·) sert à la détection de cycle négatif.

IV.2. Exemple d'application des équations de récurrence – graphe sans cycle négatif

Avant d'aller plus loin, regardons un exemple de l'application des équations de récurrence de Bellman-Ford en action (avec une vision bottom-up) pour appréhender la structure des algorithmes à venir.

Considérons le graphe d'entrée de la figure suivante à l'itération i == 0 (cas de base) :

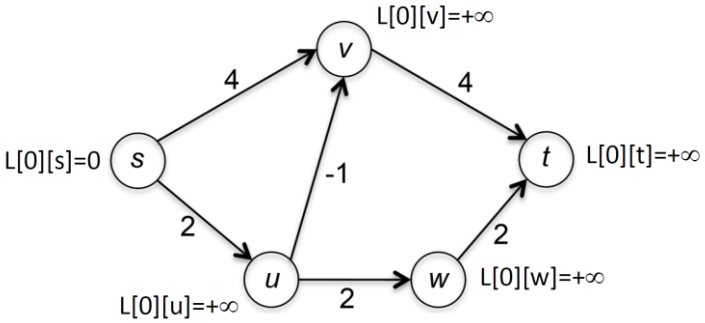


Figure 8 : Valeurs des sous-problèmes à l'itération i = 0

Chaque itération de l'algorithme évalue l'équation de récurrence en chaque sommet, en utilisant les valeurs calculées à l'itération précédente. Lors de la première itération à i == 1, la récurrence vaut :

- L[1][s] = 0 car s n'a pas d'arêtes entrantes, donc le cas 2 de la récurrence est vide ;
- L[1][u] = 2 car L[0][s] + $\ell_{s,u}$ = 2 ;
- L[1][v] = 4 car L[0][s] + $\ell_{s,v}$ =4 ;
- L[1][w] = +∞ car L[0][u] = +∞.

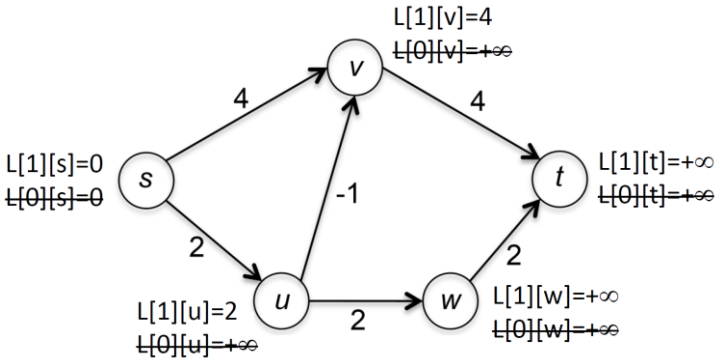


Figure 9 : Valeurs des sous-problèmes à l'itération i = 1

À l'itération suivante, s et u héritent tous deux de leurs solutions de l'itération précédente :

- La valeur en v passe à 1 car le chemin s $\rightarrow$ u $\rightarrow$ v (L[1][u] + $\ell_{u,v}$ = 1) est plus court que le chemin s $\rightarrow$ v (L[1][s] + $\ell_{s,v}$ = 4) ;
- La nouvelle valeur en w est 4 car L[1][u] + $\ell_{u,w}$ = 4 ;
- La nouvelle valeur en t est 8 car L[1][v] + $\ell_{v,t}$ = 8.

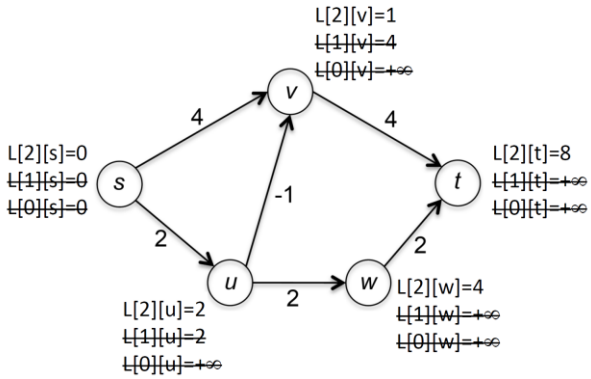


Figure 10: Valeurs des sous-problèmes à l'itération i = 2

À la troisième itération :

- La valeur en t chute à 5 car L[2][v] + $\ell_{v,t}$ = 5, ce qui est meilleur à la fois que L[2][t] = 8 et que L[2][w] + $\ell_{w,t}$ = 6 ;
- Les quatre autres sommets héritent de leurs solutions de l'itération précédente.

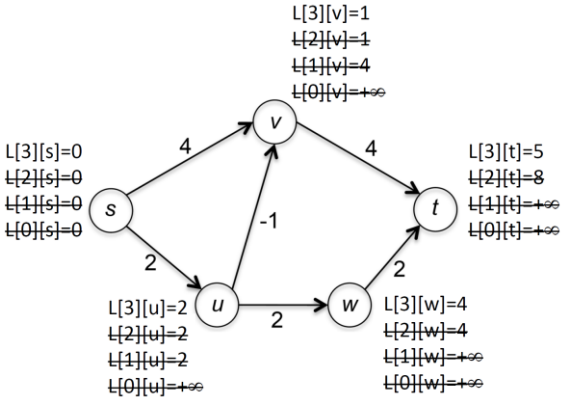


Figure 11 : Valeurs des sous-problèmes à l'itération i = 3

Rien ne change lors de la quatrième itération (le graphe n'a pas de cycle négatif donc L[i][·] = L[i-1][·]) :

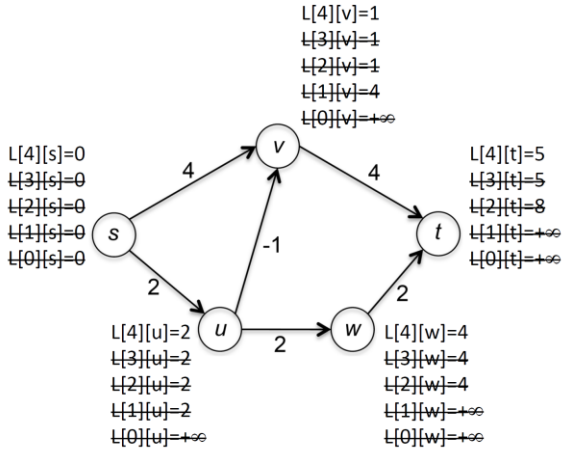


Figure 12 : Valeurs des sous-problèmes à l'itération i = 4

IV.3. Exemple d'application des équations de récurrence – graphe avec cycle négatif

Considérons maintenant le graphe ci-dessous avec un cycle négatif (u → v → w → u) qui est atteignable depuis la source, à l'itération i == 0:

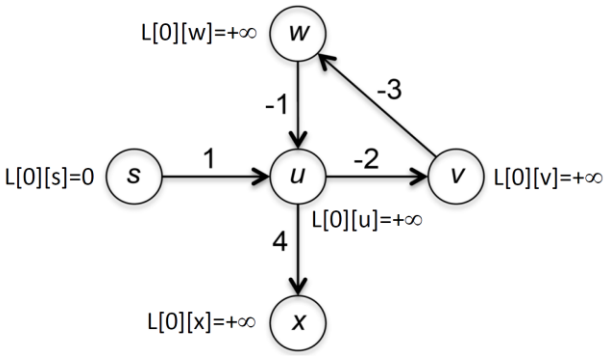


Figure 13: Valeurs des sous-problèmes à l'itération i = 0

Lors de la première itération à i == 1, la récurrence vaut :

- L[1][s] = 0 car s n'a pas d'arêtes entrantes, donc le cas 2 de la récurrence est vide ;
- L[1][u] = 1 car min {L[0][u], L[0][s] + $\ell_{s,u}$, L[0][w] + $\ell_{w,u}$} = 1 ;
- L[1][x] = +∞ car L[0][x] et L[0][u] valent tous les deux +∞ ;
- L[1][v] = +∞ car L[0][v] et L[0][u] valent tous les deux +∞ ;
- L[1][w] = +∞ car L[0][w] et L[0][v] valent tous les deux +∞ ;

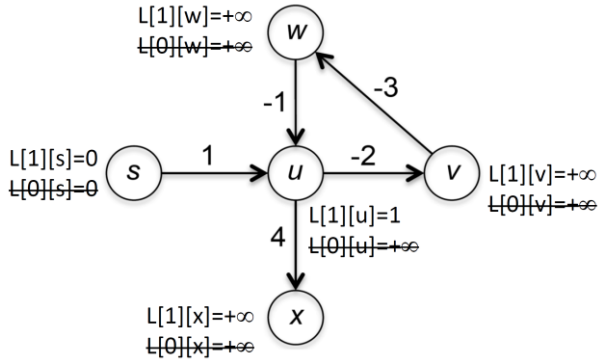


Figure 14 : Valeurs des sous-problèmes à l'itération i = 1

Lors de l'itération à i == 2, la récurrence vaut :

- L[2][u] reste à 1 car min{L[1][u], L[1][s] + $\ell_{s,u}$, L[1][w] + $\ell_{w,u}$ } = 1 ;
- L[2][x] passe à 5 car min{L[1][x], L[1][u] + $\ell_{u,x}$} = 5 ;
- L[2][v] passe à -1 car min{L[1][v], L[1][u] + $\ell_{u,v}$} = -1 ;
- L[2][w] reste à +∞ car L[1][w] et {L[1][v] + $\ell_{v,w}$} valent +∞

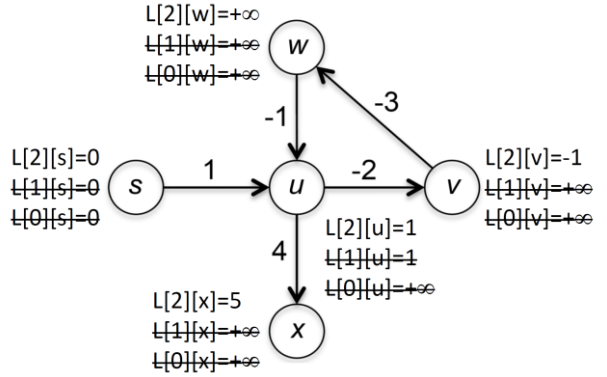


Figure 15 : Valeurs des sous-problèmes à l'itération i = 2

Lors de l'itération à i == 3, la récurrence vaut :

- L[3][u] reste à 1 car min {L[2][u], L[2][s] + $\ell_{s,u}$, L[2][w] + $\ell_{w,u}$} = 1 ;
- L[3][x] reste à 5 car min{L[2][x], L[2][u] + $\ell_{u,x}$} = 5 ;
- L[3][v] reste à -1 car min{L[2][v], L[2][u] + $\ell_{u,v}$} = -1 ;
- L[3][w] passe à -4 car min{L[2][w], L[2][v] + $\ell_{v,w}$} = -4.

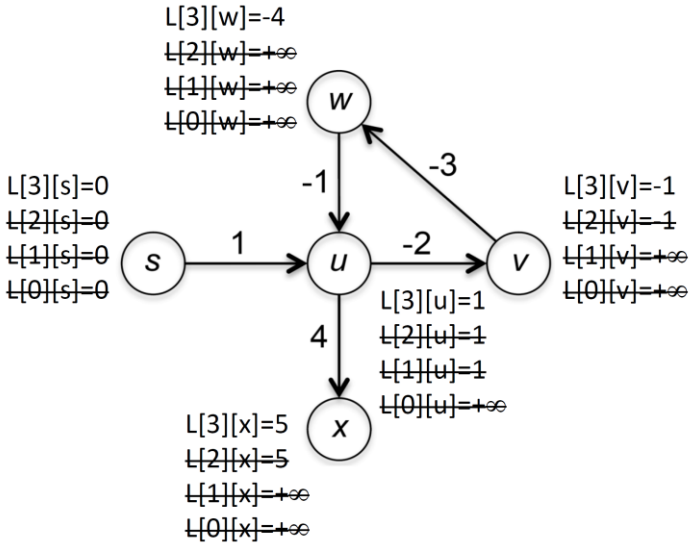


Figure 16 : Valeurs des sous-problèmes à l'itération i = 3

Lors de l'itération à i == 4, la récurrence vaut :

- L[4][u] passe à -5 car min {L[3][u], L[3][s] + $\ell_{s,u}$, L[3][w] + $\ell_{w,u}$} = -5 ;
- L[4][x] reste à 5 car min{L[3][x], L[3][u] + $\ell_{u,x}$} = 5 ;
- L[4][v] reste à -1 car min{L[3][v], L[3][u] + $\ell_{u,v}$} = -1 ;
- L[4][w] reste à -4 car min{L[3][w], L[3][v] + $\ell_{v,w}$} = -4.

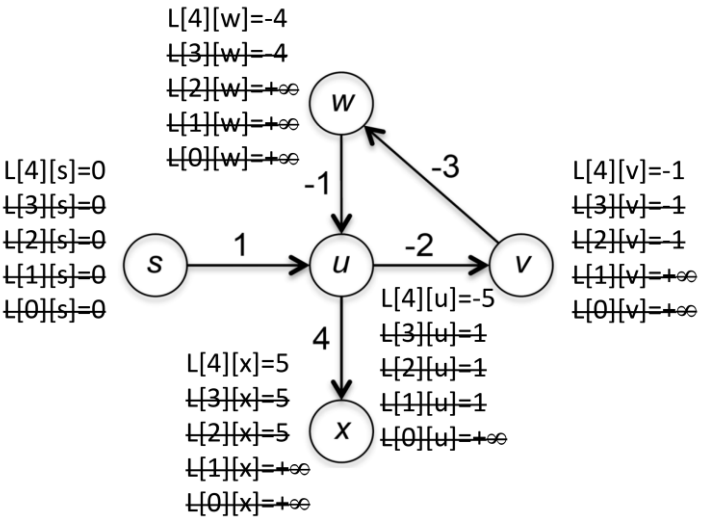


Figure 17 : Valeurs des sous-problèmes à l'itération i = 4

Sans cycle négatif, on pourrait s'arrêter à i == 4. Mais la présence d'un cycle négatif atteignable depuis la source va modifier les valeurs à la prochaine itération.

Lors de l'itération à i == 5, la récurrence vaut :

- L[5][x] passe à -1 car min{L[4][x], L[4][u] + $\ell_{u,x}$} = -1 ;
- L[5][v] passe à -7 car min{L[4][v], L[4][u] + $\ell_{u,v}$} = -7 ;
- Les autres valeurs ne changent pas.

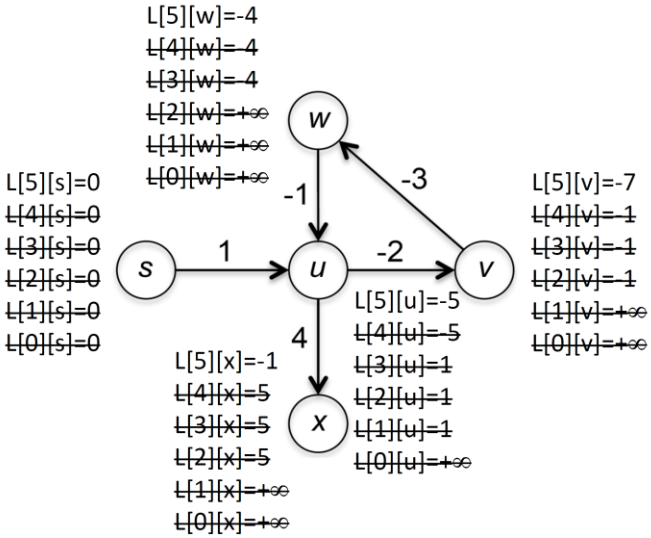


Figure 18 : Valeurs des sous-problèmes à l'itération i = 5

À ce moment, on peut renvoyer qu'un cycle négatif a été détecté car il y a eu un changement entre i == (n-1) et i == n.

Puisque le cycle négatif est atteignable depuis la source, le graphe ne se stabilisera pas. À l'itération à i == 6 :

- L[6][u] passe à -5 car min {L[5][u], L[5][s] + $\ell_{s,u}$, L[5][w] + $\ell_{w,u}$} = -5 ;
- L[6][x] reste à -1 car min{L[5][x], L[5][u] + $\ell_{u,x}$} = -1 ;
- L[6][v] reste à -7 car min{L[5][v], L[5][u] + $\ell_{u,v}$} = -7 ;
- L[6][w] passe à -10 car min{L[5][w], L[5][v] + $\ell_{v,w}$} = -10.

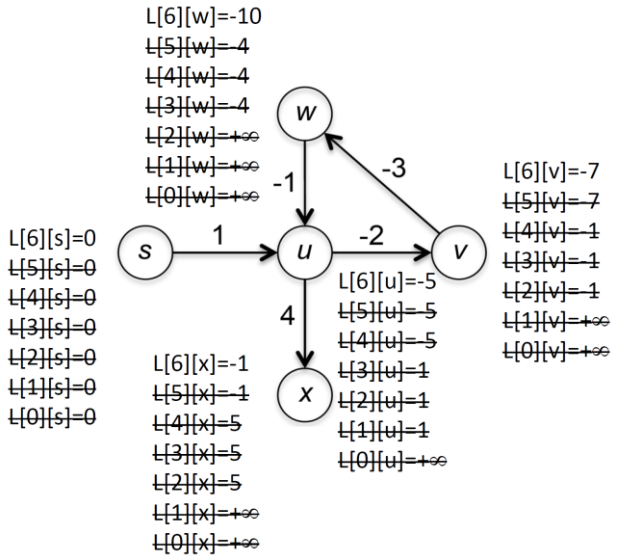


Figure 19 : Valeurs des sous-problèmes à l'itération i = 6

Ainsi, L[6][w] évolue encore, ce qui va faire évoluer par la suite les autres valeurs. Ce graphe ne se stabilisera pas car le cycle négatif u $\rightarrow$ v $\rightarrow$ w $\rightarrow$ u est atteignable depuis la source.

IV.4. Schéma de récursion

Dans les exemples précédents, nous avons surtout raisonné « par lots » : on part des cas de base i = 0, puis on calcule successivement toutes les valeurs pour i = 1, puis i = 2, etc. C'est la vision « bottom-up » naturelle d'un algorithme itératif comme Bellman-Ford.

Pour comprendre précisément les dépendances entre sous-problèmes, il est toutefois utile de changer temporairement de point de vue et d'adopter une lecture « top-down » de la récurrence : on fixe un sous-problème cible (n, v) et on « déplie » la relation de récurrence en un arbre de récursion qui montre de quels sous-problèmes plus petits dépend sa valeur. On part alors du haut (le problème que l'on cherche à résoudre) et l'on descend vers les cas de base (0, ·). Cette représentation met en évidence les sous-problèmes recalculés plusieurs fois en l'absence de mémoïsation, et elle explique naturellement pourquoi l'approche « top-down » avec mémoïsation et l'approche « bottom-up » aboutissent aux mêmes valeurs.

À partir d'un sous-problème (i, v), la récurrence fait dépendre $L_{i,v}$ de (i-1, v) (cas n°1) et de tous les (i-1, w) tels que (w, v) $\in$ E (cas 2, choix du dernier saut).

Donc l'arbre de récursion partant de (i, v) descend en diminuant i jusqu'à 0, et à chaque niveau, il peut se ramifier vers plusieurs prédécesseurs w de v. Visuellement, on peut représenter un nœud (i, v) pointant vers (i-1, v) et vers tous les (i-1, w) entrants dans v.

Le schéma de récursion sur un exemple de graphe est donné sur la figure 20 ci-dessous. Le graphe contient 4 sommets et on démarre à i = 3 (pour alléger le schéma, on considère qu'on sait qu'il n'y a pas de cycle négatif). On cherche uniquement la distance s $\rightarrow$ v :

- La notation P[3][V] signifie qu'on cherche la longueur du plus court chemin de S à V dans G comportant au plus 3 arêtes. L[3][V] est la valeur optimale associée à ce sous-problème ;
- Dans le cas n°1 (branches de gauche), à partir des cas de base, on remonte la valeur L[i-1][x] où x $\in$ V ;
- Dans les sous-cas n°2 (branches de droite), à partir des cas de base, on remonte les valeurs L[i-1][w] + $\ell_{w,x}$ pour tous les x que (w, x) $\in$ E.
- Les valeurs L[i][x] prennent le minimum des valeurs remontées.

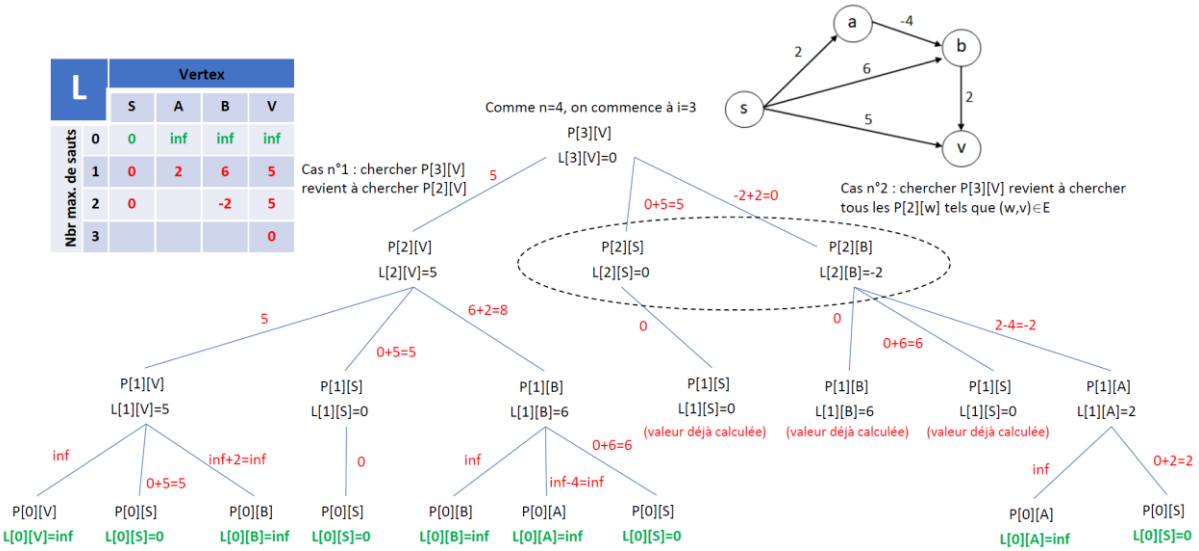


Figure 20 : Exemple de récursion sur un graphe

V) ALGORITHMES DE PROGRAMMATION DYNAMIQUE

V.1. Algorithme top-down

Algorithme top-down pour le calcul des valeurs optimales

Entrée : G = (V, E), sommet source s ∈ V, longueurs ℓ_e pour chaque arête e ∈ E.
Sortie : dictionnaire des distances dist{} (depuis la source s) ou « Cycle négatif »

```
L := {}                 # Dictionnaire de mémoïsation
dist := {}              # Dictionnaire des distances
n := |V|

rec_opt_val_BellmanFord (i, v) :
    # Utilise la mémoïsation
    Si (i, v) est dans L :
    |   Retourner L[(i, v)]

    # Cas de base
    Si i == 0 :
    |   Si v == s :
    |   |   L[(i, v)] := 0
    |   Sinon :
    |   |   L[(i, v)] := +∞
    |   Retourner L[(i, v)]

    # Cas n°1
    val_opt := rec_opt_val_BellmanFord(i - 1, v)

    # Sous-cas du cas n°2
    Pour chaque arête (w, v) entrante dans v :
    |   candidat := rec_opt_val_BellmanFord(i - 1, w) + ℓ_{w,v}
    |   val_opt := min(val_opt, candidat)

    L[(i, v)] := val_opt
    Retourner L[(i, v)]

# Calcul des distance optimale s → v, pour tous les v ∈ V
Pour chaque v dans V :
|   dist[v] := rec_opt_val_BellmanFord(n-1, v)

# Appel pour détecter les cycles négatifs atteignables depuis s
cycle_negatif := « Faux »
Pour chaque v dans V :
|   test := rec_opt_val_BellmanFord(n, v)
|   Si test < dist[v] :
|   |   cycle_negatif := « Vrai »
Si cycle_negatif == « Vrai » :
|   Retourner « Cycle négatif »
Sinon :
|   Retourner dist
```

V.2. Complexité de l'algorithme top-down

Un sous-problème est entièrement déterminé par deux paramètres : le budget de sauts i (nombre maximal d'arêtes utilisées) et la destination v (sommet d'arrivée).

Le budget maximal considéré est de (n+1) (où n = |V|) si on souhaite tester la présence de cycles négatifs (i ∈ {0, 1, …, n}). On a donc n·(n+1) sous-problèmes distincts, soit un nombre de sous-problèmes en $O(n^2)$.

Chaque sous-problème est calculé au plus une fois grâce à la mémoïsation. Si l'algorithme n'effectuait qu'un travail à coût constant par sous-problème, sa complexité temporelle serait donc de $O(n^2)$. Mais résoudre un sous-problème pour une destination v implique une recherche exhaustive parmi $1 + deg^-(v)$ candidats, où $deg^-(v)$ est le nombre d'arêtes entrantes en v. Comme le degré entrant d'un sommet peut être aussi grand que (n - 1), cela semble donner une borne en temps de $O(n)$ par sous-problème, et donc une borne en temps totale de $O(n^3)$.

Mais concentrons-nous sur une itération pour une valeur fixée de i. Le travail total effectué sur l'ensemble des itérations pour cette valeur de i est proportionnel à :

$$\sum_{v \in V} \left(1 + deg^-(v)\right) = n + \underbrace{\sum_{v \in V} deg^-(v)}_{=|E|=m}$$

La somme des degrés entrants est égale au nombre d'arêtes. Pour s'en convaincre, imaginez retirer toutes les arêtes du graphe d'entrée puis les rajouter une à une. Chaque nouvelle arête augmente de 1 le nombre total d'arêtes, et augmente aussi de 1 le degré entrant d'exactement un sommet (l'extrémité d'arrivée, ou « tête », de cette arête).

Ainsi, le travail total effectué à chaque itération pour une valeur fixée de i est O(n + m). Comme il y a au plus n itérations de ce type, le travail total est en $O(n \cdot (n+m)) = O(n \cdot m + n^2)$. Dans les cas fréquents où $m \geq n$, on obtient une borne de temps d'exécution totale de $O(m \cdot n)$. Dans les graphes clairsemés, où m est linéaire ou quasi linéaire en n (m = O(n)), cela donne $O(n^2)$.

Dans le dictionnaire, on stocke au maximum n·(n+1) valeurs, soit $O(n^2)$ valeurs. La pile de récursion a une profondeur maximale de O(n). L'espace total est donc dominé par le dictionnaire.

V.3. Algorithme bottom-up

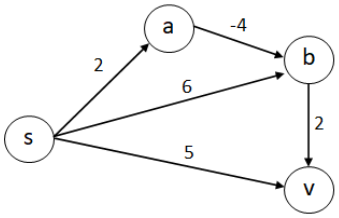
L'algorithme bottom-up remplit progressivement les valeurs $L_{i,v}$ par budgets croissants i, en partant des cas de base.

Algorithme bottom-up pour le calcul des valeurs optimales

Entrée : G = (V, E), sommet source s ∈ V, longueurs ℓ_e pour chaque arête e ∈ E.
Sortie : dictionnaire des distances dist{} (depuis la source s) ou « Cycle négatif »

```
L := {}                  # Dictionnaire de mémoïsation
dist := {}               # Dictionnaire des distances
n := |V|

opt_val_BellmanFord() :
    # Cas de base
    Pour chaque u dans V :
        L[(0, u)] := +∞
    L[(0, s)] := 0

    Pour i allant de 1 à n :
        Pour chaque u dans V :
            # Cas n°1
            L[(i, u)] := L[(i - 1, u)]

            # Sous-cas du cas n°2
            Pour chaque arête (w, u) ∈ E entrante dans u :
                L[(i, u)] := min {L[(i, u)], L[(i - 1,w)] + ℓ_{w,u}}

    # Distances
    Pour chaque u dans V :
        dist[u] := L[(n-1, u)]

    # Détection de cycle négatif atteignable depuis s
    cycle_negatif := « Faux »

    Pour chaque u dans V :
        Si L[(n, u)] < L[(n-1, u)] :
            cycle_negatif := « Vrai »

    Si cycle_negatif == « Vrai » :
        Retourner « Cycle négatif »
    Sinon :
        Retourner dist
```

Voici deux exemples de tables remplies avec l'algorithme top-down et bottom-up dans le cas où la distance recherchée est uniquement s → v :



Top-down

L		Vertex			
		S	A	B	V
Nbr max. de sauts	0	0	inf	inf	inf
	1	0	2	6	5
	2	0		-2	5
	3				0

Bottom-up

L		Vertex			
		S	A	B	V
Nbr max. de sauts	0	0	inf	inf	inf
	1	0	2	6	5
	2	0	2	-2	5
	3	0	2	-2	0

Figure 21 : Tables des valeurs optimales top-down (gauche) et bottom-up (droite)

Comme d'habitude, l'algorithme top-down ne calcule que les sous-problèmes nécessaires à la distance demandée. Si on ne cherche que la distance vers une destination v, le top-down peut éviter de remplir des états inutiles ; si on veut toutes les distances dist(s,v) pour tout v, alors top-down et bottom-up calculent en pratique une grande partie des états.

V.4. Complexité de l'algorithme bottom-up

Le bottom-up calcule systématiquement tous les sous-problèmes (i, v) pour tous les i et v. Il y a donc $O(n^2)$ problèmes à calculer, et comme nous l'avons vu précédemment la complexité temporelle est de $O(n \cdot m)$ car chaque sous-problème examine les arêtes entrantes. La complexité spatiale est de $O(n^2)$.

On pourrait facilement réduire l'espace à O(n) en ne gardant que deux lignes de la table (ligne i-1 et ligne i) car ce sont uniquement ces lignes qui sont nécessaires pour calculer les équations de récurrence. Mais cette optimisation a un inconvénient : on perd la capacité de reconstruire le chemin optimal directement à partir de la table, puisqu'on ne conserve plus l'historique complet des valeurs $L_{i,v}$. Pour pallier ce problème, on peut maintenir un dictionnaire de prédécesseurs `pred{}` qu'on met à jour à chaque fois qu'on améliore la distance vers un sommet v (voir les méthodes de reconstruction en page 21).

Au lieu de calculer tous les $L_{i,v}$ pour i = {0, 1, .., n} et pour chaque v ∈ V, à chaque itération on calcule pour chaque sommet v :

$$L_v = min \begin{cases} L_v & (cas\ n°1) \\ min_{(w,v)\in E}\{ L_w + \ell_{w,v}\} & (cas\ n°\ 2) \end{cases}$$

Le premier terme correspond au cas n°1 (on garde la valeur précédente), le second au cas n°2 (on améliore via un prédécesseur).

Algorithme bottom-up optimisé en mémoire avec sauvegarde des prédécesseurs

Entrée : G = (V, E), sommet source s ∈ V, longueurs ℓ_e pour chaque arête e ∈ E.
Sortie : dictionnaire des distances dist{} (depuis la source s) et dictionnaire des prédécesseurs pred ou « Cycle négatif »

```
dist := {}              # Dictionnaire des distances
pred := {}              # Dictionnaire des prédécesseurs
n := |V|

opt_val_BellmanFord() :
    # Cas de base (i == 0)
    Pour chaque u dans V :
        Si u == s :
            dist[u] := 0
        Sinon :
            dist[u] := +∞
        pred[u] := None

    # Itérations principales (n-1 itérations pour les distances)
    Pour i allant de 1 à n-1 :
        Pour chaque u dans V :
            Pour chaque arête (w, u) entrante dans u :
                Si dist[w] + ℓ_{w,u} < dist[u] :
                    dist[u] := dist[w] + ℓ_{w,u}
                    pred[u] := w

    # Détection de cycle négatif (n-ième itération) atteignable depuis s
    cycle_negatif := « Faux »
    Pour chaque u dans V :
        Pour chaque arête (w, u) entrante dans u :
            Si dist[w] + ℓ_{w,u} < dist[u] :
                cycle_negatif := « Vrai »

    Si cycle_negatif == « Vrai » :
        Retourner « Cycle négatif »
    Sinon :
        Retourner dist, pred
```

VI) ALGORITHME DE RECONSTRUCTION

VI.1. Principe et algorithme de reconstruction

L'objectif est ici de reconstruire un plus court chemin (ou, plus généralement, un « arbre des prédécesseurs ») depuis s vers un sommet v, à partir des informations calculées par la programmation dynamique.

On peut considérer deux approches : la reconstruction à partir de la table L[i][v] et la reconstruction via un tableau de prédécesseurs.

Première approche : Reconstruction à partir de la table L[i][v]

On part de (i, v) = (n - 1, v) et on remonte :

- Si L[i][v] == L[i-1][v], alors la solution optimale pour le budget i n'utilise pas l'arête supplémentaire : on remplace i par i-1.
- Sinon, on cherche un prédécesseur w tel que (w, v) ∈ E et

$$L[i][v] = L[i-1][w] + \ell_{w,v}$$

  … puis ajoute l'arête (w, v) à la reconstruction, puis on remplace v ← w et i ← i-1
- On s'arrête quand v = s (chemin reconstruit).

Dans l'approche top-down avec mémoïsation, les valeurs L[i][v] sont stockées dans un dictionnaire indexé par les paires (i, v). Lorsqu'on calcule effectivement L[i][v] à l'aide de la récurrence, on est amené à évaluer L[i-1][v] (cas n°1) ainsi que L[i-1][w] pour tous les prédécesseurs w tels que (w, v) ∈ E (cas n°2).

Par conséquent, si on a déjà calculé L[n-1][v] en top-down, toutes les clés (i', x) nécessaires à la reconstruction d'un plus court chemin vers ce même sommet v ont en principe été rencontrées et mémorisées au passage.

Pour rendre la reconstruction robuste, on peut soit appeler la fonction récursive mémoïsée au moment de la reconstruction (ce qui calcule la valeur si elle manque), soit, plus simplement, mémoriser dès le calcul de chaque état (i, v) le « choix » gagnant (héritage v ou prédécesseur w) : la reconstruction devient alors un simple parcours de pointeurs, sans dépendre de la présence de toutes les valeurs intermédiaires dans le dictionnaire.

Algorithme de reconstruction

Entrée : G = (V, E), table L = {…}, source s et destination v.
Sortie : Liste d'arêtes (ou sommets) du chemin optimum s → v.

```
Reconstruction_BellmanFord (L, s, v) :
    Si L[(n-1, v)] == +∞ :
        Retourner « destination inatteignable »

    chemin = []        # Chemin à reconstruire
    i := n - 1
    s_cur := v         # Sommet courant

    Tant que s_cur ≠ s et i > 0 :
        Si L[(i-1, s_cur)] existe et L[(i, s_cur)] == L[(i-1, s_cur)] :
            i := i - 1
        Sinon :
            Trouver un prédécesseur w tel que : { (w, s_cur) ∈ E
                                                { L[(i, s_cur)] == L[(i − 1, w)] + ℓ_{w,s_cur}
            Ajouter (w, s_cur) à chemin
            s_cur := w
            i := i - 1
    Retourner le chemin inversé
```

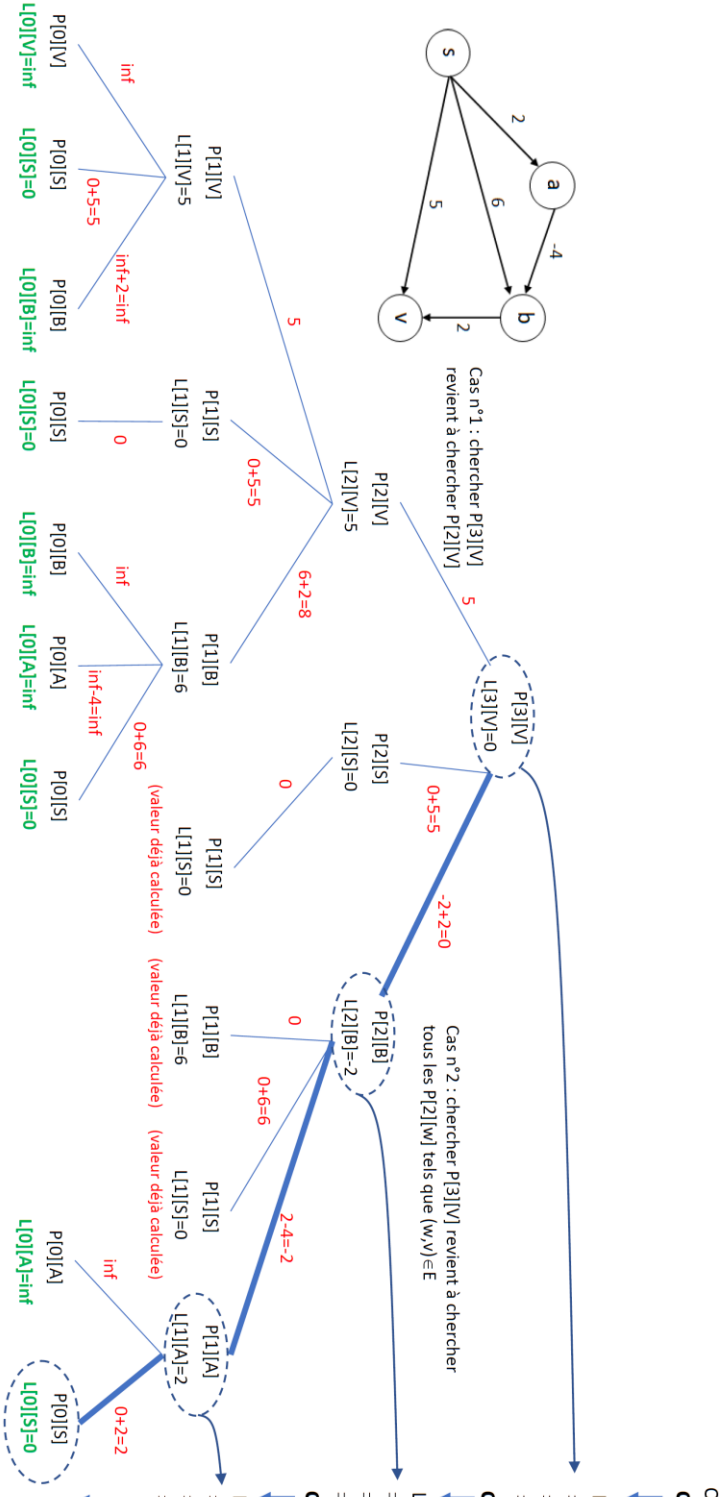


Figure 22 : Illustration du principe de reconstruction à l'aide de la table des valeurs optimales

Deuxième approche : reconstruction via un tableau de prédécesseurs.
Pendant les itérations, on maintient la liste `pred_v[]`, qui enregistre les sommets précédents choisis quand on améliore `dist_v`. Si l'algorithme termine sans cycle négatif, suivre `pred_v` depuis v jusqu'à s reconstruit un plus court chemin.

Cette méthode de reconstruction est particulièrement adaptée dans le cas où on cherche à économiser de l'espace mémoire en ne gardant en mémoire que les deux lignes de la table (ligne i-1 et ligne i).

Algorithme de reconstruction via un tableau de prédécesseurs

Entrée : dictionnaire des prédécesseurs pred{}, source s, destination v.
Sortie : Liste d'arêtes (ou sommets) du chemin optimum s $\rightarrow$ v.

```
Reconstruction_BellmanFord (pred, s, v) :
    # Vérifier que v est atteignable
    Si pred[v] == None et v ≠ s :
        Retourner « Destination inatteignable »

    # Reconstruction en remontant les prédécesseurs
    chemin := [v]
    sommet_courant := v

    Tant que sommet_courant ≠ s :
        sommet_courant := pred[sommet_courant]
        Ajouter sommet_courant à chemin

    Retourner chemin inversé
```

VI.2. Complexité finale

Pendant la reconstruction, avec la table L, on décrémente i au plus (n-1) fois et on suit au plus (n-1) arêtes. La complexité temporelle est donc en O(n) pour reconstruire un chemin vers une destination donnée.

La complexité temporelle totale (calcul + reconstruction) est donc de $O(n \cdot m) + O(n) = O(n \cdot m)$.