COURS : PROGRAMMATION DYNAMIQUE = ALGORITHME DE FLOYD-WARSHALL=

Notre dernière étude vise à généraliser le calcul des plus courts chemins dans un graphe en passant du cas « une source vers tous » au cas « toutes les paires ». Dans de nombreuses applications (par exemple un service d'itinéraires), on ne connaît pas à l'avance le sommet de départ : il faut pouvoir répondre pour n'importe quelle origine et n'importe quelle destination. On conserve les mêmes subtilités que précédemment, en autorisant des longueurs d'arêtes négatives et en exigeant, le cas échéant, la détection d'un cycle négatif plutôt que la production de distances ambiguës.

I) PLUS COURTS CHEMINS ENTRE TOUTES LES PAIRES DE SOMMETS

I.1. Définition du problème

Pourquoi se contenter de calculer les distances de plus court chemin à partir d'un seul sommet source ? Par exemple, un algorithme de calcul d'itinéraires routiers doit pouvoir prendre en charge n'importe quel point de départ ; cela correspond au problème des plus courts chemins entre toutes les paires de sommets. Nous continuons d'autoriser, dans le graphe d'entrée, des arêtes de longueur négative ainsi que des cycles négatifs.

Problème des plus courts chemins entre toutes les paires de sommets

Entrée : Un graphe orienté G = (V, E) avec n sommets et m arêtes, et une longueur réelle ℓ_e pour chaque arête e ∈ E.
Sortie : l'un des résultats suivants :

- la distance de plus court chemin dist(v, w) pour chaque paire ordonnée de sommets v, w ∈ V ; ou
- une déclaration indiquant que G contient un cycle négatif.

Il n'y a pas de sommet source dans le problème des plus courts chemins toutes paires. Dans le cas n°1, l'algorithme doit produire n^2 nombres.

Sachant que la complexité d'une approche exhaustive est de O(n·n!) dans le cas d'une source unique (voir le cours sur Bellman-Ford), elle passe ici en $O(n^2 \cdot n!)$, ce qui est prohibitif. On va donc chercher une approche qui exploite une sous-structure optimale et des sous-problèmes recouvrants, comme dans l'étude de Bellman-Ford.

I.2. Réduction aux plus courts chemins à source unique

Une approche naturelle consiste à répéter une sous-routine qui résout le problème du plus court chemin depuis une source unique (comme l'algorithme de Bellman-Ford).

Un seul appel à la sous-routine de l'algorithme de Bellman-Ford calcule les distances de plus court chemin depuis un sommet s vers tous les sommets du graphe (soit n nombres au total, sur les n^2 requis). En appelant la sous-routine une fois pour chacun des n choix possibles de s, on obtient les distances de plus court chemin pour toutes les origines et toutes les destinations possibles. La complexité temporelle est alors de $O(n^2 \cdot m)$.

La borne de temps d'exécution $O(n^2 \cdot m)$ est particulièrement problématique dans les graphes denses. Par exemple, si $m = O(n^2)$, le temps d'exécution devient quartique en n, ce qui est encore trop élevé.

II) L'ALGORITHME DE FLOYD-WARSHALL

II.1. Sous-problèmes

Trouver le bon découpage en sous-problèmes pour une solution par programmation dynamique à un problème sur les graphes peut être délicat. L'idée ingénieuse qui sous-tend les sous-problèmes de l'algorithme de Bellman-Ford pour le problème du plus court chemin à source unique consiste à toujours travailler avec le graphe d'entrée original et à imposer une contrainte artificielle sur le nombre d'arêtes autorisées dans la solution d'un sous-

problème. Ce « budget » d'arêtes sert alors de mesure de la taille du sous-problème, et un préfixe d'une solution optimale d'un sous-problème peut être interprété comme une solution à un sous-problème plus petit (avec la même origine mais une destination différente).

L'idée maîtresse de l'algorithme de Floyd–Warshall est d'aller encore plus loin en restreignant artificiellement l'identité des sommets autorisés à apparaître dans une solution.

Pour définir les sous-problèmes, considérons un graphe d'entrée G = (V, E) et attribuons arbitrairement à ses sommets les noms 1, 2, …, n (où n=|V|). Les sous-problèmes sont alors indexés par des préfixes {1, 2, …, k} de l'ensemble des sommets (k servant à la fois de mesure de la taille du sous-problème) ainsi que par une origine v et une destination w.

Sous-problèmes de l'algorithme de Floyd-Warshall

Calculer $L_{k,v,w}$, la longueur minimale d'un chemin dans le graphe d'entrée G qui :

- commence en v et se termine en w ;
- n'utilise comme sommets **internes** que des sommets appartenant à {1, 2, …, k} ;
- et (hypothèse) ne contient pas de cycle négatif (on retourne « Cycle négatif détecté » sinon)

(Si aucun chemin de ce type n'existe, on définit $L_{k,v,w} = +\infty$)

(Pour chaque $k \in \{0, 1, 2, \ldots, n\}$ et $v, w \in V$)

Il y a $(n + 1)\cdot n\cdot n = O(n^3)$ sous-problèmes. Le lot des plus grands sous-problèmes (avec k = n) correspond au problème initial. Pour une origine v et une destination w fixées, l'ensemble des chemins autorisés s'élargit lorsque k augmente, et par conséquent $L_{k,v,w}$ ne peut que décroître quand k croît.

Considérons par exemple le graphe de la figure 1 avec pour l'origine 1 et la destination 5 et les sous-problèmes correspondant aux valeurs successives de la longueur du préfixe k :

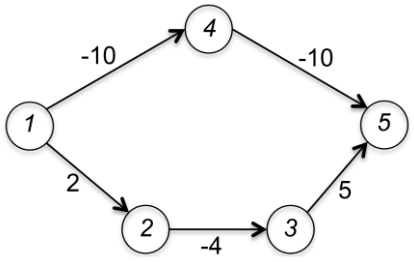


Figure 1 : Exemple de graphe

- Quand k vaut 0, 1 ou 2, il n'existe aucun chemin de 1 à 5 tel que tout sommet interne appartienne au préfixe {1, 2, …, k}, et la solution du sous-problème est $+\infty$.
- Quand k = 3, le chemin $1 \to 2 \to 3 \to 5$ devient l'unique chemin admissible ; sa longueur est 2 + (−4) + 5 = 3. Le chemin en deux sauts est disqualifié parce qu'il inclut le sommet 4 comme sommet interne. Le chemin en trois sauts est admissible même si le sommet 5 n'appartient pas au préfixe {1, 2, 3} car en tant que destination, ce sommet bénéficie d'une exemption et n'est pas considéré comme sommet interne.
- Quand k = 4 (ou davantage), la solution du sous-problème est la longueur du véritable plus court chemin $1 \to 4 \to 5$, qui vaut -20.

Nous verrons que l'intérêt de définir les sous-problèmes de cette manière est qu'il n'existe que deux candidats possibles pour une solution optimale d'un sous-problème, selon qu'elle utilise ou non le dernier sommet autorisé k (à l'inverse, dans l'algorithme de Bellman-Ford, le nombre de solutions candidates pour un sous-problème dépend du degré entrant du sommet de destination). Cela conduit à un algorithme de programmation dynamique qui n'effectue que O(1) travail par sous-problème et qui est donc plus rapide que n exécutions de l'algorithme de Bellman-Ford (avec un temps d'exécution $O(n^3)$ plutôt que $O(n^2 \cdot m)$).

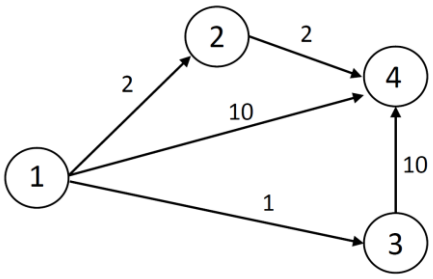
II.2. Sous-structure optimale

Considérons un graphe d'entrée G = (V, E) dont les sommets sont étiquetés de 1 à n, et fixons un sous-problème, défini par un sommet d'origine v, un sommet de destination w et une longueur de préfixe k ∈ {1, 2, …, n}.

Supposons que P soit un chemin de v à w sans cycle, dont tous les sommets internes appartiennent à {1, 2, …, k}, et qu'il s'agisse en outre d'un plus court chemin parmi ceux-ci. À quoi doit-il ressembler ? Le dernier sommet autorisé k apparaît soit comme sommet interne de P, soit il n'y apparaît pas.

Cas n°1 : Le sommet k n'est pas un sommet interne à P.
Dans ce cas, le chemin P peut immédiatement être interprété comme une solution au sous-problème plus petit avec une longueur de préfixe k - 1, toujours avec l'origine v et la destination w.

Exemple : Prenons le graphe orienté à 4 sommets V = {1, 2, 3, 4} ci-dessous, la source 1 et la destination 4 :



Considérons la solution optimale avec k = 4, c'est-à-dire dont tous les sommets internes appartiennent à {1, 2, 3, 4}. Cette solution est le chemin P = (1 → 2 → 4) qui ne contient que le sommet interne {2}.

Puisque le sommet k = 4 n'est pas un sommet interne de P, on peut interpréter P comme une solution au sous-problème plus petit avec k = 3. La solution de ce sous-problème reste la même : P = (1 → 2 → 4), dont les sommets internes sont dans {1, 2, 3}.

De nouveau, puisque le sommet k = 3 n'est pas un sommet interne de P, on peut interpréter P comme une solution au sous-problème plus petit avec k = 2. La solution de ce sous-problème reste la même : P = (1 → 2 → 4), dont les sommets internes sont dans {1, 2}

À ce stade, puisque le sommet k = 2 appartient à P, on ne peut plus continuer avec le cas n°1. Analysons donc maintenant le cas n°2.

Cas n°2 : Le sommet k est un sommet interne à P.

Dans ce cas, le chemin P peut être interprété comme la concaténation de deux solutions à des sous-problèmes plus petits : le préfixe P_1 de P, qui va de v à k, et le suffixe P_2 de P, qui va de k à w :

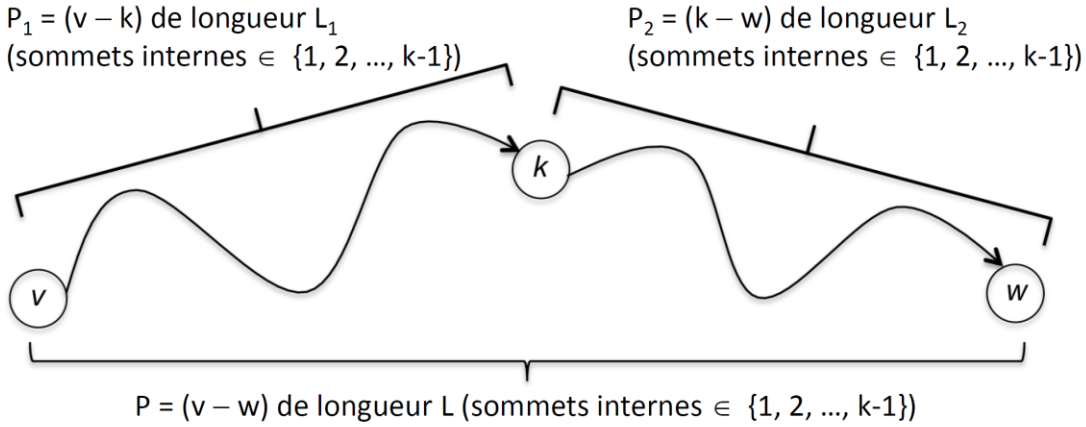
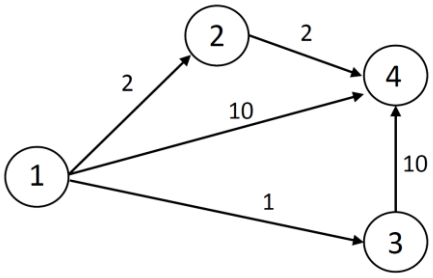


Figure 2 : Illustration du cas n°2

Le sommet k n'apparaît qu'une seule fois dans P (puisque P ne contient pas de cycle) et, par conséquent, il n'est pas un sommet interne de P_1 ni de P_2. On peut donc considérer P_1 et P_2 comme des solutions à des sous-problèmes plus petits, d'origines v et k et de destinations k et w respectivement, et dont tous les sommets internes appartiennent à {1, 2, …, k−1}.

Ce dernier argument explique pourquoi les sous-problèmes de Floyd–Warshall, contrairement à ceux de Bellman-Ford, imposent la condition d'absence de cycle. Notons que cette approche ne fonctionnerait pas bien pour le problème du plus court chemin à source unique, car le chemin suffixe P_2 aurait un sommet d'origine incorrect.

Exemple : Reprenons le graphe orienté à 4 sommets précédent, la source étant le sommet 1 et la destination 4 :



Puisque les sommets 4 et 3 ne sont pas des sommets internes à la solution optimale P = (1 → 2 → 4), ce problème revient à chercher la solution au sous-problème P = (1 → 2 → 4) lorsque k = 2.

Lorsque k = 2, nous sommes dans le cas n°2 puisque le sommet k = 2 est utilisé comme sommet intermédiaire. Le chemin P peut donc se décomposer en deux sous-chemins :

- P1 = {1 … k} = {1 → 2}
- P2 = {k … 4} = {2 → 4}

Ces deux sous-chemins doivent avoir leurs sommets internes dans {1, 2, ..., k-1} = {1}.

II.3. Équation de récurrence sur les valeurs optimales

On note $L_{k,v,w}$ la longueur minimale d'un chemin sans cycle de v à w avec tous les sommets intermédiaires dans {1, 2, ..., k} (s'il n'existe pas de tels chemins, alors $L_{k,v,w} = +\infty$).

Récurrence sur la valeur de la solution optimale

Les cas de base sont pour k = 0 :

- $L_{0,v,v} = 0$ pour tout v ∈ V (chemin vide de longueur 0) ;
- $L_{0,v,w} = \ell_{v,w}$ (arête directe) si (v, w) ∈ E
- $L_{0,v,w} = +\infty$ si $v \neq w$ et $(v, w) \notin E$

Pour tout $k \in \{1, 2, \ldots, n\}$ et v,w ∈ V :

$$L_{k,v,w} = min \begin{cases} L_{k-1,v,w} & (cas\ n°1) \\ L_{k-1,v,k} + L_{k-1,k,w} & (cas\ n°\ 2) \end{cases}$$

Le premier terme dans le « min » correspond au cas 1 (le sommet k n'est pas utilisé comme intermédiaire). Le second terme correspond au cas 2 (le sommet k est utilisé comme intermédiaire, et le chemin se décompose en un chemin $v \to k$ et un chemin $k \to w$).

II.4. Détection d'un cycle négatif

Les entrées « diagonales » du tableau des sous-problèmes sont révélatrices de la présence d'un cycle négatif :

Détection d'un cycle négatif

Le graphe d'entrée G = (V, E) contient un cycle négatif si et seulement si, à la fin de l'algorithme de Floyd–Warshall, on a $L_{n,v,v} < 0$ pour un certain sommet v ∈ V.

La suite de cette partie est consacrée à la démonstration de ce résultat.

Si le graphe d'entrée ne contient pas de cycle négatif, alors l'algorithme Floyd–Warshall calcule correctement toutes les distances de plus court chemin et il n'existe aucun chemin d'un sommet v vers lui-même qui soit plus court que le chemin vide (de longueur 0). Ainsi, à la fin de l'algorithme, on a $L_{n,v,v} = 0$ pour tout v ∈ V.

Si un cycle négatif est présent, les « distances » au sens « min » sur tous les chemins (avec cycles autorisés) peuvent tendre vers -∞, donc l'algorithme n'est pas tenu de renvoyer des valeurs « correctes » en tant que distances. Cependant, on va montrer que même si un cycle négatif existe, les quantités $L_{k,v,w}$ calculées par l'algorithme pour les chemins sans cycle restent toujours inférieures ou égales à la meilleure longueur d'un chemin $v \to w$ sans cycle, dont les sommets internes sont restreints à {1, 2, …, k}. Les quantités calculées par Floyd–Warshall gardent donc malgré tout une propriété de majoration « par le bas » vis-à-vis des chemins sans cycle.

Ce résultat peut se prouver par récurrence et est important pour montrer ensuite la propriété « $L_{n,v,v} < 0$ pour un certain sommet v ∈ V si G contient un cycle négatif ».

Initialisation : si k = 0, aucun sommet interne n'est autorisé. Les seuls chemins sans cycle possibles sont :

- le chemin vide (si v = w) de longueur 0 ;
- ou l'arête directe v → w si elle existe (de longueur $\ell_{v,w}$)
- ou rien (donc +∞)

Donc $L_{0,v,w}$ est exactement la longueur minimale d'un chemin sans cycle avec sommets internes dans ∅.

Hérédité : supposons que cela soit vrai pour (k - 1) et montrons que cela est vrai également pour k. Prenons un chemin P quelconque v → w, sans cycle, dont les sommets internes sont dans {1 ,…, k}.

Dans le cas où k n'est pas un sommet interne de P (cas n°1 de la sous-structure optimale), alors tous les sommets internes de P sont dans {1, …, k-1}. Donc P est admissible au niveau (k-1) et donc on a $L_{k,v,w} \leq \ longueur\ (P)$

Dans le cas où k est un sommet interne de P (cas n°2), comme P est sans cycle, le sommet k apparaît une seule fois sur P. On peut donc écrire P = P_1 + P_2 où P_1 = (v → k) et P_2 = (k → w), tous les deux sans cycle, et dont les sommets sont dans {1, …, k-1}. Par hypothèse de récurrence on a :

$$L_{k-1,v,k} \leq \ longueur\ (P_1)\ et\ L_{k-1,k,w} \leq \ longueur\ (P_2)$$

En sommant :

$$L_{k-1,v,k} + L_{k-1,k,w} \leq \ longueur\ (P)$$

Or, d'après l'équation de récurrence de Floyd-Warshall, on a :

$$L_{k,v,w} = min \begin{cases} L_{k-1,v,w} \\ \\ L_{k-1,v,k} \ + \ L_{k-1,k,w} \end{cases}$$

Donc puisque $L_{k,v,w} \leq \ L_{k-1,v,k} + L_{k-1,k,w}$, on obtient $L_{k,v,w} \leq \ longueur\ (P)$.

En conclusion, dans tous les cas on a : $L_{k,v,w} \leq \ longueur\ (P)$.

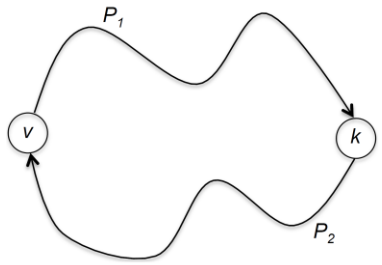
L'inégalité ci-dessus montre que les valeurs renvoyées par l'algorithme de Floyd–Warshall ne sont jamais « trop grandes » par rapport aux meilleurs chemins simples autorisés, même si un cycle négatif est présent.

Nous allons maintenant montrer la propriété « $L_{n,v,v} < 0$ pour un certain sommet v ∈ V si G contient un cycle négatif ».

Supposons que G contienne un cycle négatif. Ce cycle peut être un cycle « complexe » où l'on autorise des répétitions de sommets. Mais si ce cycle fait une boucle et qu'il repasse par un même sommet, on a en réalité fait « deux boucles » imbriquées et si la somme des deux

est négative, au moins une des deux boucles est déjà négative. On peut donc extraire d'un cycle négatif « complexe » un cycle négatif plus « simple » (qui utilise strictement moins de répétitions, ou au moins moins de sommets/arêtes). En répétant cette idée tant qu'il reste une répétition interne, on finit (car on retire des portions et la longueur décroît) par obtenir un cycle négatif « simple » qui ne répète aucun sommet, sauf le premier/dernier. Cela implique que G possède forcément un cycle négatif ne comportant aucun sommet répété, à l'exception de son sommet de départ et de son sommet d'arrivée.

Notons C le cycle « simple », choisi arbitrairement. Supposons que le sommet k du cycle simple C ait l'étiquette la plus grande. Soit $v \neq k$ un autre sommet de C :



Les deux « côtés » P_1 et P_2 du cycle sont des chemins sans cycle de v à k et de k à v, dont les sommets internes sont restreints à {1, 2, …, k−1}. D'après ce que nous avons vu précédemment, on a :

$$L_{k-1,v,k} \leq \ longueur\ (P_1)\ et\ L_{k-1,k,v} \leq \ longueur\ (P_2)$$

Puisque :

$$L_{k,v,v} \leq L_{k-1,v,k} + L_{k-1,k,v} \leq \ longueur\ (C) < 0$$

D'après la récurrence de Floyd-Warshall, on a finalement la valeur finale : $L_{n,v,v} < 0$.

Remarque : la « diagonale négative » $L_{n,v,v} < 0$ est l'expression du fait qu'un chemin $v \rightarrow v$ plus court que le chemin vide (longueur 0) ne peut exister que s'il contient un cycle de longueur négative.

III) SOUS-PROBLÈMES ET COMPLEXITÉ

III.1. Définition des sous-problèmes

Rappelons ici les sous-problèmes :

Sous-problèmes de l'algorithme de Floyd-Warshall

Calculer $L_{k,v,w}$, la longueur minimale d'un chemin sans cycle de v à w dans G avec tous les sommets intermédiaires dans {1, 2, ..., k}. Si un tel chemin n'existe pas, $L_{k,v,w} = +\infty$.

(Pour chaque k = 0, 1, 2, ..., n et chaque v, w ∈ V)

Le nombre total de sous-problèmes est $(n + 1)\cdot n\cdot n = O(n^3)$.

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

Prenons l'exemple du graphe en figure 3 et étudions le cas particulier où la source est le sommet s et la destination le sommet t. Les sommets sont étiquetés de la manière suivante : s = 1, v = 2, u = 3, w = 4 et t = 5.

Avec une vision de type bottom-up, au début de l'algorithme k = 0 (aucun sommet interne n'est autorisé) et les valeurs des cas de base sont les suivantes :

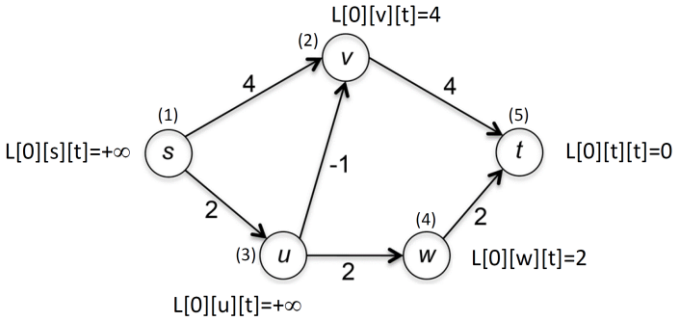


Figure 3: Valeurs des sous-problèmes pour k = 0

Pour k = 1, on autorise le sommet interne s et la récurrence donne :

- L[1][s][t] = min {L[0][s][t], L[0][s][s] + L[0][s][t]} = +∞ (on ne peut pas aller de s → t en prenant le sommet interne s ;
- L[1][v][t] reste à 4 (le chemin v → t est direct)
- L[1][u][t] = min {L[0][u][t], L[0][u][s] + L[0][s][t]} = +∞ (on ne peut pas aller de u → t en prenant le sommet interne s ;
- L[1][w][t] reste à 2 (le chemin w → t est direct)
- L[1][t][t] reste à 0 (chemin vide de longueur 0)

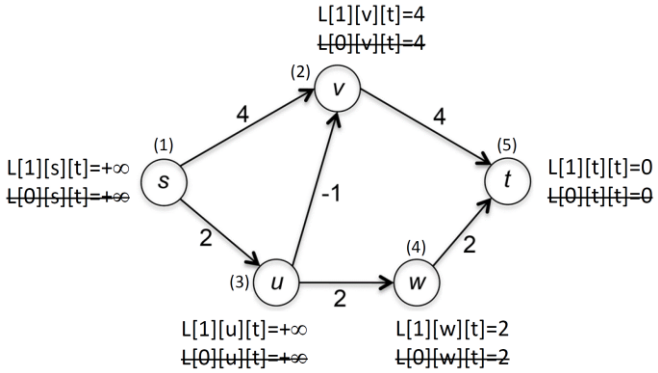


Figure 4: Valeurs des sous-problèmes pour k = 1

Pour k = 2, on autorise les sommets internes {s, v}. Les valeurs qui évoluent sont :

- L[2][s][t] = min {L[1][s][t], L[1][s][v] + L[1][v][t]} = min {+∞, 4+4} = 8 ;
- L[2][u][t] = min {L[1][u][t], L[1][u][v] + L[1][v][t]} = min {+∞, -1+4} = 3

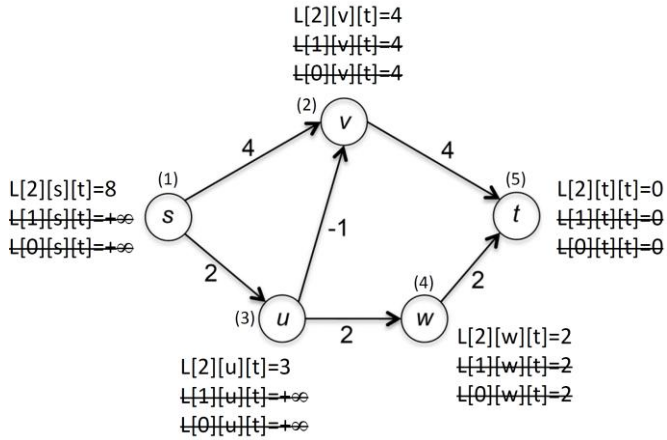


Figure 5 : Valeurs des sous-problèmes pour k = 2

Pour k = 3, on autorise les sommets internes {s, v, u}. La valeur qui évolue est L[3][s][t] :

- L[3][s][t] = min {L[2][s][t], L[2][s][u] + L[2][u][t]} = min {8, 2+3} = 5 ;

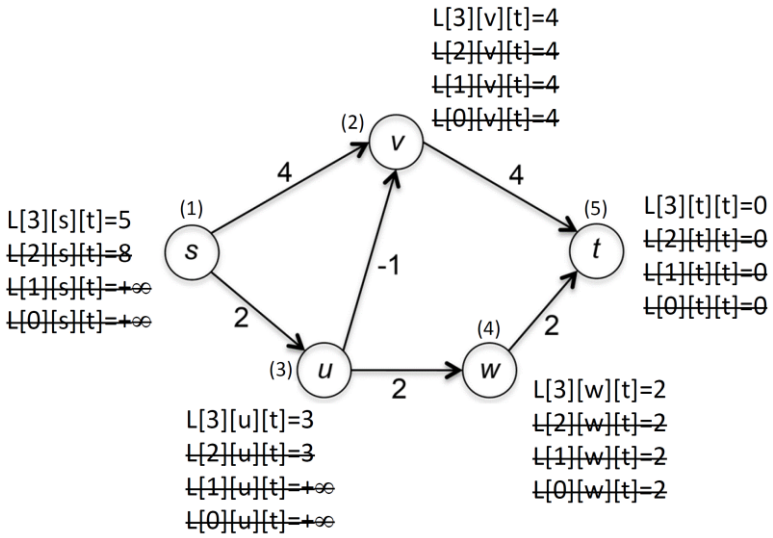


Figure 6 : Valeurs des sous-problèmes pour k = 3

Pour k = 4, on autorise les sommets internes {s, v, u, w}. Cela n'améliore rien depuis le sommet u car le chemin {u → v → t} est plus court que le nouveau chemin autorisé {u → w → t} et on garde les mêmes valeurs.

Pour k = 5, on autorise l'ensemble des sommets internes {s, v, u, w, t} mais passer par le sommet t n'apporte aucun bénéfice et donc les valeurs restent identiques.

Le résultat final est donc obtenu à k = 3, autorisant les sommets internes {s, v, u}. La distance minimale pour aller de s → t est donc de 5, en suivant le chemin optimal {s → u → v → t}.

III.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, et étudions le cas particulier où la source est le sommet s et la destination le sommet v. Les sommets sont étiquetés de la manière suivante : s = 1, u = 2, x = 3, w = 4 et v = 5. À l'itération k == 0, on a les valeurs suivantes :

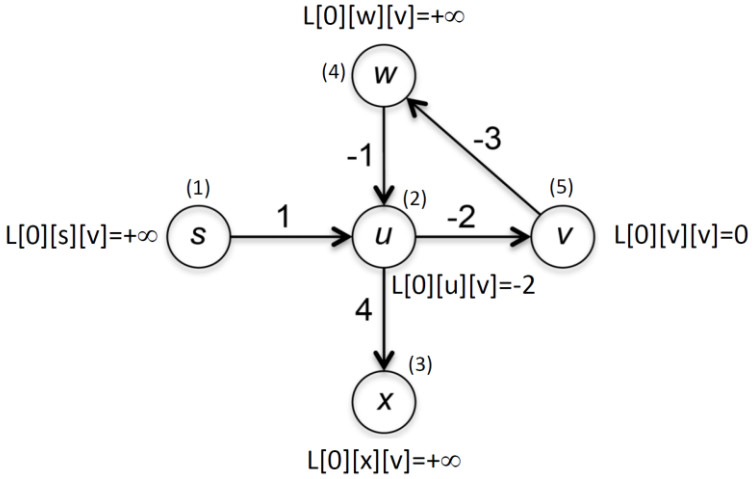


Figure 7: Valeurs des sous-problèmes pour k = 0

Pour k = 1, on autorise le sommet interne s et les valeurs resteront les mêmes car autoriser le sommet s ne permet pas d'améliorer les valeurs.

Pour k = 2, on autorise les sommets internes {s, u}, ce qui va permettre d'améliorer la distance des chemins s $\rightarrow$ v et w $\rightarrow$ v :

- L[2][s][v] = min {L[1][s][v], L[1][s][u] + L[1][u][v]} = min {+∞, 1 - 2} = -1
- L[2][w][v] = min {L[1][w][v], L[1][w][u] + L[1][u][v]} = min {+∞, -1 - 2} = -3

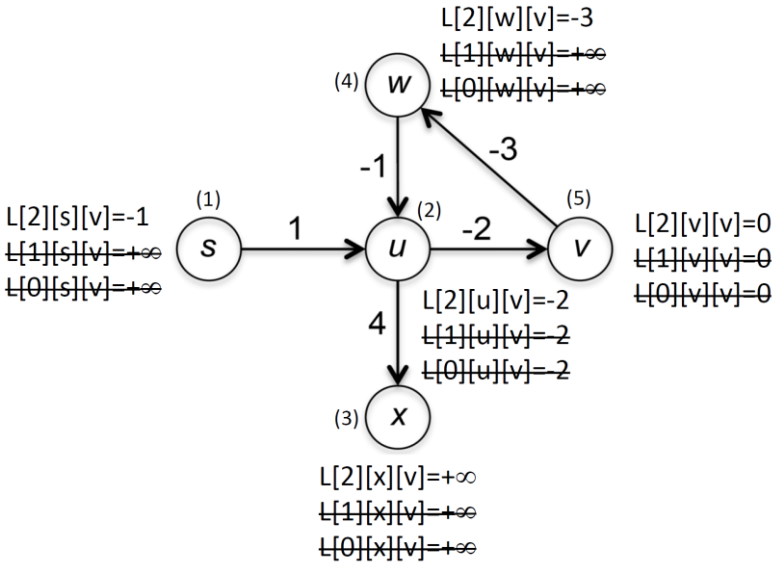


Figure 8 : Valeurs des sous-problèmes pour k = 2

Pour k = 3, on autorise en plus le sommet interne x et rien ne changera pour les valeurs.

Pour k = 4, on autorise maintenant les sommets internes {s, u, x, w}. On commence à faire apparaître la diagonale négative L[4][v][v] :

- L[4][v][v] = min {L[3][v][v], L[3][v][w] + L[3][w][v]} = min {0, -3 - 3} = -6

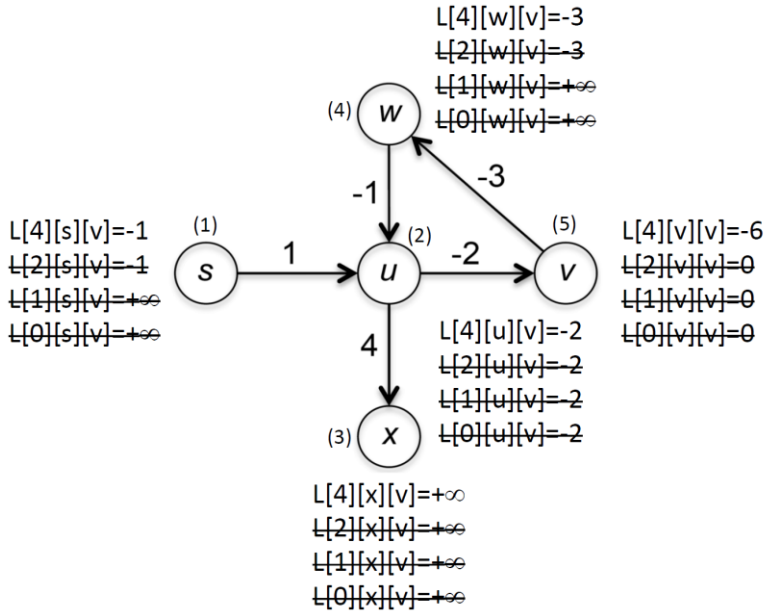


Figure 9 : Valeurs des sous-problèmes pour k = 4

Pour k = 5, on ajoute le sommet v aux sommets autorisés :

- L[5][s][v] = min {L[4][s][v], L[4][s][v] + L[4][v][v]} = min {-1, -1 - 6} = -7
- L[5][u][v] = min {L[4][u][v], L[4][u][v] + L[4][v][v]} = min {-2, -2 - 6} = -8
- L[5][w][v] = min {L[4][w][v], L[4][w][v] + L[4][v][v]} = min {-3, -3 - 6} = -9
- L[5][v][v] = min {L[4][v][v], L[4][v][v] + L[4][v][v]} = min {-6, -6 - 6} = -12

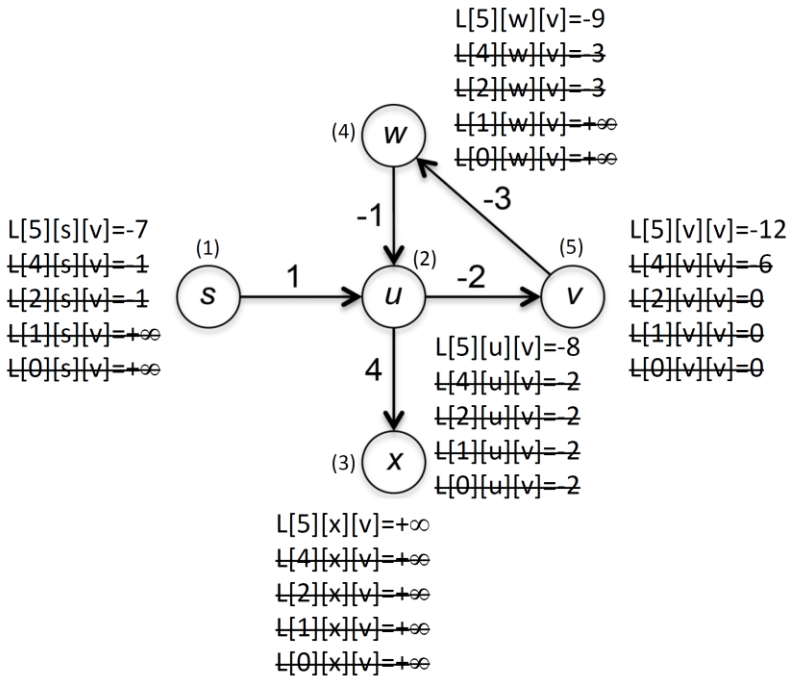


Figure 10: Valeurs des sous-problèmes pour k = 5

Le fait que L[5][v][v] soit négatif signifie qu'on a détecté un cycle négatif.

III.4. Remarque sur la détection d'un cycle négatif

Dans l'exemple précédent, on a vu que le cycle négatif a été détecté dès l'itération k = 4 (L[4][v][v] < 0). La preuve que nous avons faite en page 7-8 n'oblige pas à aller jusqu'à k = n si un cycle négatif est détecté à k.

L'algorithme standard de type « bottom-up » vise à calculer toutes les distances (quand il n'y a pas de cycle négatif), et à fournir un test uniforme. On exécute donc systématiquement toutes les itérations k = 1, …, n, puis on teste la diagonale à la fin. Mais pour la détection seule, on peut arrêter dès qu'une diagonale devient négative.

On peut donc inclure une détection de cycle négatif au fur et à mesure que l'algorithme de type « top-down » calcule les diagonales et en détecte une strictement négative. Mais on ne peut pas conclure « pas de cycle négatif » tant qu'on n'a pas forcé le calcul de toutes les diagonales pertinentes (typiquement L[n][v][v] pour tout v).

IV) ALGORITHMES DE PROGRAMMATION DYNAMIQUE

IV.1. Algorithme top-down

Algorithme top-down pour le calcul des valeurs optimales

Entrée : G = (V, E), avec V = {1, 2, …, n}, longueurs ℓ_e pour chaque arête e ∈ E.
Sortie : dictionnaire des distances dist{} ou « Cycle négatif »

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

rec_opt_val_FloydWarshall(k, v, w) :
    # Utilise la mémoïsation
    Si (k, v, w) est dans L :
        Retourner L[(k, v, w)]

    # Cas de base (k == 0)
    Si k == 0 :
        Si v == w :
            L[(k, v, w)] := 0
        Sinon si (v, w) est une arête de G :
            L[(k, v, w)] := ℓ_vw
        Sinon :
            L[(k, v, w)] := +∞
        Retourner L[(k, v, w)]

    # Cas récursif
    cas1 := rec_opt_val_FloydWarshall(k-1,v,w)
    cas2 := rec_opt_val_FloydWarshall(k-1,v,k) + rec_opt_val_FloydWarshall(k-1,k,w)
    L[(k, v, w)] := min(cas1, cas2)

    # Détection précoce d'un cycle négatif si c'est un sous-problème diagonal
    Si v == w et L[(k, v, w)] < 0 :
        Lever une exception « Cycle négatif détecté »

    Retourner L[(k, v, w)]

# Calcul des distances optimales
Pour chaque v dans V :
    Pour chaque w dans V :
        dist[(v,w)] := rec_opt_val_FloydWarshall(n, v, w)

Retourner dist
```

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

Le nombre de sous-problèmes distincts est de $(n + 1)\cdot n^2 = O(n^3)$ et le travail par sous-problème est en O(1). Donc, si l'on calcule toutes les paires (v, w), le temps est $O(n^3)$. L'espace de mémoïsation est $O(n^3)$ si l'on mémorise tous les états calculés.

Remarque : si on ne demande qu'une seule paire (v, w), le top-down peut éviter de calculer certains états inutiles. Toutefois, dans le pire cas, l'ensemble des dépendances peut encore forcer le calcul d'une fraction importante des $O(n^3)$ états. Le bottom-up, lui, calcule systématiquement tous les états.

IV.3. Algorithme bottom-up

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

Algorithme bottom-up pour le calcul des valeurs optimales

Entrée : G = (V, E), avec V = {1, 2, …, n}, longueurs ℓ_e pour chaque arête e ∈ E.
Sortie : dictionnaire des distances dist{} ou « Cycle négatif »

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

opt_val_FloydWarshall() :
    # Cas de base (k = 0)
    Pour v allant de 1 à n :
        Pour w allant de 1 à n :
            Si v == w :
                L[(0, v, w)] := 0
            Sinon si (v, w) est une arête de G :
                L[(0, v, w)] := ℓvw
            Sinon :
                L[(0, v, w)] := +∞

    # Résoudre systématiquement tous les sous-problèmes
    Pour k allant de 1 à n :    # préfixe des sommets autorisés
        Pour v allant de 1 à n :   # origine
            Pour w allant de 1 à n :   # destination
                # Utiliser la récurrence
                L[(k, v, w)] := min (L[(k - 1, v, w)], L[(k - 1, v, k)] + L[(k - 1, k, w)])

    # Vérifier la présence d'un cycle négatif
    Pour v allant de 1 à n :
        Si L[(n, v, v)] < 0 :
            Retourner « Cycle négatif détecté »

    dist := L[(n, v, w)] pour tout v, w ∈ G
    Retourner dist
```

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

L'algorithme bottom-up calcule systématiquement tous les sous-problèmes (k, v, w) pour tous les k, v et w. Il y a $O(n^3)$ sous-problèmes à calculer, et chaque sous-problème s'exécute en O(1) (une comparaison et deux additions).

La complexité temporelle est donc $O(n^3)$ et la complexité spatiale $O(n^3)$.

On peut réduire l'espace à $O(n^2)$ en ne gardant que deux « tranches » du tableau (tranche k-1 et tranche k) car ce sont uniquement ces tranches qui sont nécessaires pour calculer les équations de récurrence.

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

$$L_{v,w} = min \begin{cases} L_{v,w} & (cas\ n°1) \\ L_{v,k} + L_{k,w} & (cas\ n°\ 2) \end{cases}$$

Le premier terme correspond au cas n°1 (on garde la valeur précédente), le second au cas n°2 (on améliore via le sommet k comme nouvel intermédiaire).

Algorithme bottom-up optimisé en mémoire

Entrée : G = (V, E), avec V = {1, 2, …, n}, longueurs ℓ_e pour chaque arête e ∈ E.
Sortie : dictionnaire des distances dist{} ou « Cycle négatif »

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

opt_val_FloydWarshall() :
    # Initialisation (k = 0)
    Pour v allant de 1 à n :
        Pour w allant de 1 à n :
            Si v == w :
                dist[(v, w)] := 0
            Sinon si (v, w) ∈ E :
                dist[(v, w)] := ℓvw
            Sinon :
                dist[(v, w)] := +∞
    # Itérations principales
    Pour k allant de 1 à n :
        Pour v allant de 1 à n :
            Pour w allant de 1 à n :
                dist[(v, w)] := min(dist[(v, w)], dist[(v, k)] + dist[(k, w)])
    # Détection de cycle négatif
    Pour v allant de 1 à n :
        Si dist[(v, v)] < 0 :
            Retourner « Cycle négatif détecté »

    Retourner dist
```

V) ALGORITHME DE RECONSTRUCTION

V.1. Principe et algorithme de reconstruction

L'objectif est de reconstruire un plus court chemin depuis un sommet v vers un sommet w, à partir des informations calculées par la programmation dynamique.

On part de (k, v, w) = (n, v, w) et on remonte :

- Si L[k][v][w] == L[k-1][v][w], alors le sommet k n'est pas utilisé comme intermédiaire : on passe à k - 1 ;
- Sinon, L[k][v][w] == L[k-1][v][k] + L[k-1][k][w], et le sommet k est sur le chemin optimal. On reconstruit récursivement le chemin v $\rightarrow$ k puis le chemin k $\rightarrow$ w.
- On s'arrête quand k = 0 (chemin direct ou pas de chemin).

Algorithme de reconstruction

Entrée : Dictionnaire L = {…}, origine v et destination w.
Sortie : Liste d'arêtes (ou sommets) du chemin optimum v $\rightarrow$ w.

```
Reconstruction_FloydWarshall(L, v, w) :
    # Cas trivial : origine == destination
    Si v == w :
        Retourner [v]
    # Cas où il n'y a pas de chemin
    Si L[(n, v, w)] == +∞ :
        Lever une exception « Pas de chemin »
    # Fonction récursive
    rec_FloydWarshall(k, v, w) :
        # Cas trivial si k == 0 : arête directe
        Si k == 0 :
            Retourner [v, w]
        # Cas n°1 : k n'est pas un sommet intermédiaire
        Si  L[(k, v, w)] == L[(k - 1, v, w)] :
            Retourner  rec_FloydWarshall(k-1, v, w)
        # Cas n°2
        chemin1 := rec_FloydWarshall(k-1, v, k)
        chemin2 := rec_FloydWarshall(k-1, k, w)
        # Concaténer en évitant de dupliquer k
        Retourner chemin1 + chemin2[1:]
    n := |V|
    Retourner rec_FloydWarshall(n, v, w)
```

V.2. Complexité finale

Pendant la reconstruction, on descend de k = n à k = 0, et à chaque niveau on ajoute un sommet intermédiaire. La complexité de reconstruction d'un chemin est donc O(n) dans le pire cas, et la reconstruction de l'ensemble des chemins est de $O(n^3)$ car il y a n^2 chemins de longueur O(n). Dans tous les cas, la complexité totale finale est en $O(n^3)$.