COURS : INTRODUCTION À LA PROGRAMMATION DYNAMIQUE

Vous avez vu en première année deux manières de programmer des algorithmes : la méthode « diviser pour régner » et celle des algorithmes gloutons. La première méthode consiste à diviser un problème en sous-problèmes indépendants qu'on résout puis qu'on combine, tandis que les algorithmes gloutons construisent une solution étape par étape en faisant à chaque fois un choix localement optimal sans revenir en arrière. Ces méthodes ne couvrent pas tous les problèmes de calcul que l'on peut rencontrer.

Nous allons introduire dans ce cours une troisième méthode : la programmation dynamique. C'est une technique particulièrement puissante, car elle conduit souvent à des solutions efficaces.

Ici, nous allons étudier un algorithme faisant partie des « grands classiques » afin d'apprendre le fonctionnement de cette méthode.

I) PROBLÈME DE L'ENSEMBLE INDÉPENDANT PONDÉRÉ

Nous allons concevoir depuis zéro un algorithme pour un problème de calcul concret et délicat, ce qui nous forcera à développer un certain nombre de nouvelles idées. Une fois que nous aurons résolu le problème, nous identifierons les ingrédients de notre solution qui illustrent les principes généraux de la programmation dynamique.

I.1. Définition du problème

Pour décrire le problème, soit G = (V, E) un graphe non orienté. Un ensemble indépendant de G est un sous-ensemble S ⊆ V de sommets mutuellement non adjacents : pour tous v, w dans S, on a (v, w) ∉ E. Autrement dit, un ensemble indépendant ne contient pas les deux extrémités d'une même arête de G.

Par exemple, si les sommets représentent des personnes et les arêtes des paires de personnes qui ne s'aiment pas, les ensembles indépendants correspondent aux groupes de personnes qui s'entendent tous bien. Ou encore, si les sommets représentent des cours que vous envisagez de suivre et qu'il y a une arête entre chaque paire de cours en conflit, les ensembles indépendants correspondent aux emplois du temps réalisables (en supposant que vous ne puissiez pas être à deux endroits à la fois).

Par exemple, le graphe de gauche ci-dessous possède six ensembles indépendants : l'ensemble vide et les cinq singletons. Celui de droite possède les mêmes ensembles, plus cinq autres ensembles indépendants de dimension 2 : {A, C}, {B, D}, {C, E}, {D, A} et {E, B}.

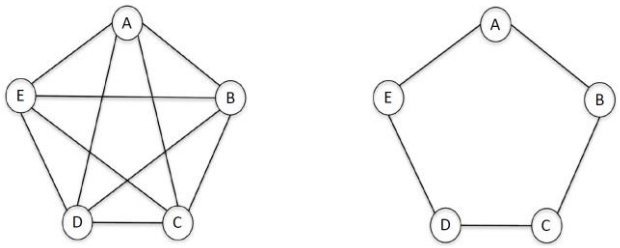


Figure 1 : Exemples de graphe

Le problème de l'ensemble indépendant pondéré (Weighted Independent Set – WIS) s'énonce ainsi :

Problème : Ensemble indépendant pondéré (WIS)

Entrée : Un graphe non orienté G = (V, E) et un poids non négatif w_v pour chaque sommet v ∈ V.

Sortie : Un ensemble indépendant S ⊆ V de G ayant la somme de poids des sommets $\sum_{v \in S} w_v$ aussi grande que possible.

Une solution optimale au problème de l'ensemble indépendant pondéré (WIS) est appelée ensemble indépendant de poids maximal (Maximum WIS – MWIS). Par exemple, si les sommets représentent des cours, les poids des sommets représentent le nombre d'unités, et les arêtes représentent les conflits entre les cours, alors le MWIS correspond à l'emploi du temps réalisable avec la charge la plus lourde (en unités).

Le problème de l'ensemble indépendant pondéré est difficile même dans le cas simple des chemins. Par exemple, une instance du problème pourrait ressembler à ceci (avec les sommets étiquetés par leurs poids) :

Figure 2 : Problème WIS sous forme d'un chemin

Ce graphe possède 8 ensembles indépendants : {∅}, {A}, {B}, {C}, {D}, {A, C}, {A, D} et {B, D}. Le dernier de ces ensembles possède le plus grand poids total, égal à 8. Le nombre d'ensembles indépendants d'un graphe chemin croît exponentiellement avec le nombre de sommets, donc il n'y a aucun espoir de résoudre le problème par recherche exhaustive, sauf pour les toutes petites instances.

I.2. Que donnerait un algorithme glouton sur ce cas ?

Pour de nombreux problèmes de calcul, les algorithmes gloutons sont un excellent point de départ. De tels algorithmes sont généralement faciles à imaginer, et même lorsqu'ils ne parviennent pas à résoudre le problème (ce qui arrive souvent), la manière dont ils échouent peut aider à mieux comprendre les subtilités du problème.

Pour le problème WIS, l'algorithme glouton le plus naturel est sans doute celui-ci : on parcourt une seule fois les sommets, du meilleur (poids le plus élevé) au pire (poids le plus faible), en ajoutant un sommet à la solution courante tant qu'il n'entre pas en conflit avec un sommet déjà choisi.

Dans l'exemple de la figure 2, la première itération de l'algorithme glouton choisirait donc le sommet de poids maximum 5, c'est-à-dire « C ». Puisque les sommets avec les poids juste au-dessous de 5 (« D » et « B ») ne sont pas indépendants de « C », l'algorithme choisirait ensuite d'ajouter « A » à « C » et renverrait donc l'ensemble {C, A} dont le poids total est 6, et qui n'est pas la solution optimale.

I.3. Approche « diviser pour régner »

La conception d'algorithmes « diviser pour régner » vaut toujours la peine d'être essayée pour les problèmes où il existe un moyen naturel de découper l'entrée en sous-problèmes plus petits.

Pour le problème WIS avec un graphe chemin G = (V, E) comme entrée, l'approche naturelle pourrait être :

Problème de l'ensemble indépendant pondéré : approche « Diviser pour mieux régner »

G_1 := Première moitié de G
G_2 := Seconde moitié de G
S_1 := Résoudre de manière récursive le problème sur G1
S_2 := Résoudre de manière récursive le problème sur G2
Combiner S_1, S_2 en une solution S pour G

Le problème va se poser dans l'étape de combinaison. Le premier et le deuxième appel récursif remontent les solutions optimales « B » et « C », mais leur combinaison ne forme pas un ensemble indépendant :

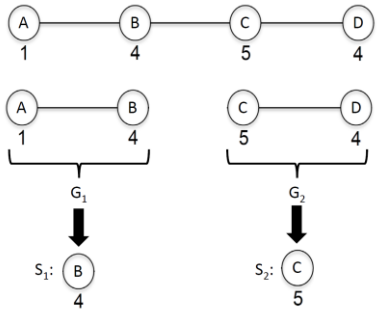


Figure 3 : Méthode "Diviser pour mieux régner" sur le problème WIS

On peut désamorcer un conflit à la frontière lorsque le graphe d'entrée n'a que quatre sommets mais quand il en a des centaines ou des milliers, cela devient très compliqué.

II) ALGORITHME LINÉAIRE POUR LE PROBLÈME

II.1. Sous-structure optimale

Idéalement, une solution optimale doit être construite d'une manière déterminée à partir de solutions optimales de sous-problèmes plus petits, réduisant ainsi le champ des candidats à un nombre gérable.

Plus concrètement, prenons G = (V, E) le graphe de type chemin à n sommets, avec les arêtes (v_1, v_2), (v_2, v_3), …, (v_{n-2}, v_{n-1}), (v_{n-1}, v_n) et un poids non négatif w_i pour chaque sommet $v_i \in V$. Supposons que $n \geq 2$; sinon, la réponse est évidente :

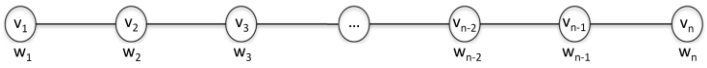
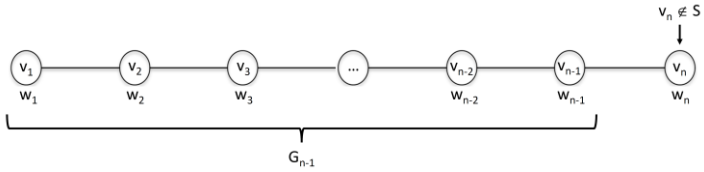


Figure 4 : Graphe G = (V, E) de type chemin à n sommets

Supposons connu un ensemble indépendant pondéré $S \subseteq V$, qui est une solution optimale du problème et dont le poids total est W. Deux cas sont possibles : S ne contient pas le dernier sommet v_n, soit il le contient. Examinons ces deux cas.

Cas n°1 : $v_n \notin S$. Supposons que la solution optimale S n'inclue pas le dernier sommet v_n.

On obtient le graphe de type chemin à (n – 1) sommets G_{n-1} à partir de G en retirant le dernier sommet v_n et la dernière arête (v_{n-1}, v_n). Comme S n'inclut pas le dernier sommet de G, il ne contient que des sommets de G_{n-1} et S peut donc être considéré comme un ensemble indépendant de G_{n-1} (toujours de poids total W) :

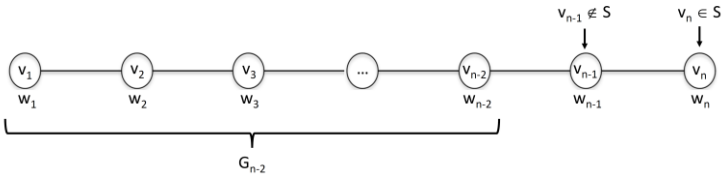


Ainsi, une fois que l'on sait qu'une solution optimale exclut le dernier sommet, on sait exactement à quoi elle ressemble : c'est la solution optimale du graphe plus petit G_{n-1}.

Si S (de poids total W) est solution optimale de G et $v_n \notin S$
$\Rightarrow$ S (de poids total W) est solution optimale de G_{n-1}

Cas n°2 : $v_n \in S$. Supposons que la solution optimale S inclue le dernier sommet v_n.

Comme S est un ensemble indépendant, S ne peut pas contenir deux sommets consécutifs du chemin, donc il exclut l'avant-dernier sommet : $v_{n-1} \notin S$. On obtient le graphe de type chemin à (n – 2) sommets G_{n-2} à partir de G en retirant les deux derniers sommets et arêtes (si n = 2, on interprète G_0 comme le graphe vide avec un poids total de 0).



Comme S contient v_n et que G_{n-2} ne le contient pas, on ne peut pas considérer S comme un ensemble indépendant de G_{n-2} et donc comme une solution optimale de G_{n-2}. Mais après avoir retiré le dernier sommet de S, on peut le faire : $S - \{v_n\}$ ne contient ni v_{n-1} ni v_n et peut donc être considéré comme un ensemble indépendant du graphe plus petit G_{n-2} (avec un poids total $W - w_n$).

Ainsi, une fois que l'on sait qu'une solution optimale inclut le dernier sommet, on sait exactement à quoi elle ressemble : c'est la solution optimale du graphe plus petit G_{n-2}, complété par le dernier sommet v_n.

Si S (de poids total W) est solution optimale de G et $v_n \in S$
$\Rightarrow$ $S - \{v_n\}$ (de poids total $W - w_n$) est solution optimale de G_{n-2}
$\Rightarrow$ S (de poids total W) est solution optimale de $G_{n-2} \cup \{v_n\}$

II.2. Équation de récurrence sur les valeurs optimales

On a isolé les deux seules possibilités pour une solution, donc celle des deux dont le poids total est le plus grand est la solution optimale. Nous avons donc une récurrence pour le poids total d'une solution optimale à notre problème :

Récurrence sur le poids de la solution optimale

Soit W_i le poids total d'une solution optimale d'un ensemble indépendant pondéré de G_i (quand i = 0, on interprète W_0 comme 0). Alors, pour tout i = 2, 3, …, n :

$$W_i = \max\left\{ \underbrace{W_{i-1}}_{\text{cas 1}}, \underbrace{W_{i-2} + w_i}_{\text{cas 2}} \right\}$$

II.3. Approche récursive naïve

Nous avons réduit le champ des possibles à seulement deux candidats pour la solution optimale. Dans le pseudocode suivant, on va essayer les deux options et retourner la meilleure. Les graphes G_{n-1} et G_{n-2} sont définis comme précédemment :

Algorithme récursif pour le problème du meilleur ensemble indépendant pondéré

Entrée : Un graphe de type chemin G avec l'ensemble de sommets $\{v_1, v_2, \ldots, v_n\}$ et un poids non négatif w_i pour chaque sommet v_i.

Sortie : Un ensemble indépendant de poids maximal de G.

Si n = 0 **alors** : # cas de base
    Retourner $\{\emptyset\}$
Si n = 1 **alors** : # cas de base
    Retourner $\{v_1\}$

Récursion lorsque n $\geq$2

S_1 := calculer récursivement la solution optimale de G_{n-1}
S_2 := calculer récursivement la solution optimale de G_{n-2}

Retourner S_1 ou $S_2 \cup \{v_n\}$, selon lequel a le poids le plus élevé

Le schéma de récursion ressemble à la figure 5 (on cherche S = MWIS de [A, B, C, D]) :

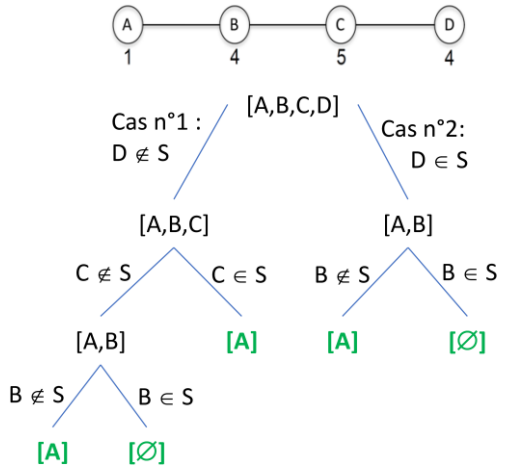


Figure 5 : Exemple d'arbre de récursion pour la recherche d'un MWIS (en vert : les cas de base)

Le schéma de récursion ressemble à celui des algorithmes de type diviser pour régner en temps O(n log n) comme « MergeSort », avec deux appels récursifs suivis d'une étape de combinaison simple. Mais il y a une grande différence : l'algorithme « MergeSort » écarte la moitié de l'entrée avant de lancer la récursion, alors que notre algorithme récursif n'élimine qu'un ou deux sommets (sur des milliers voire des millions).

Les deux algorithmes ont des arbres de récursion avec un facteur de branchement égal à 2. Le premier possède environ $\log_2 n$ niveaux, la récursion s'arrête quand il arrive à des sous-tableaux de taille 1 (cas de base). Chaque appel tout en bas de l'arbre de récursion correspond donc à trier un sous-tableau qui ne contient qu'un seul élément. Il y a donc environ n appels au niveau du bas de l'arbre, donc un temps en O(n log n).

Dans le deuxième, chaque niveau de récursivité ne peut enlever qu'au plus 2 sommets. Si on descend de k niveaux dans l'arbre de récursion, on aura enlevé au plus 2·k sommets. Donc pour partir de n sommets et arriver à 0 ou 1 sommet et donc aux cas de base, il faut au moins descendre jusqu'au niveau n/2. Tous les nœuds jusqu'au niveau (n/2 – 1) sont donc des nœuds internes qui se ramifient encore. Le nombre de nœuds peut donc aller jusqu'à $2^{n/2}$ au niveau n/2. Pour cet algorithme récursif, le temps d'exécution est exponentiel : il faut au moins un nombre d'appels récursifs proportionnel à $2^{n/2}$. Autrement dit, le temps explose comme $2^{n/2} = (\sqrt{2})^n = 1{,}414^n$ quand n grandit.

Plus exactement, la récurrence du nombre d'appels T(n) est de la forme :

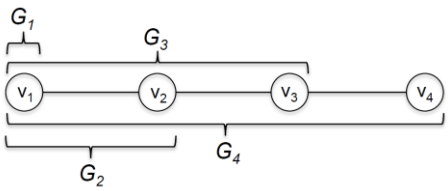
$$T(n) = T(n-1) + T(n-2) + O(1)$$

… qui est exactement la récurrence de Fibonacci. La solution est $T(n) = O(\varphi^n)$, avec $\varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}618$.

II.4. Récursion avec un cache mémoire (mémoïsation)

L'algorithme précédent n'est donc pas meilleur qu'une recherche exhaustive, mais on peut se demander, parmi l'ensemble des appels récursifs, combien de graphes d'entrée distincts sont réellement examinés.

Soit un, soit deux sommets et arêtes sont retirés à la fin du graphe. Ainsi, un invariant tout au long de la récursion est que chaque appel récursif reçoit en entrée un certain préfixe G_i comme graphe d'entrée, où G_i désigne les i premiers sommets et les (i – 1) premières arêtes du graphe d'entrée original (et G_0 désigne le graphe vide).



Il n'existe que (n + 1) graphes de ce type (G_0, G_1, G_2, …, G_n), où n est le nombre de sommets du graphe d'entrée. Par conséquent, seulement (n + 1) sous-problèmes distincts sont réellement résolus parmi l'exponentiel nombre de différents appels récursifs.

Cela montre que le temps d'exécution exponentiel de l'algorithme récursif provient uniquement de la redondance des sous-problèmes à traiter. La première fois que l'on résout un sous-problème, l'idée est donc d'enregistrer le résultat dans un cache une bonne fois pour toutes. Ainsi, si l'on rencontre le même sous-problème plus tard, on peut simplement retrouver sa solution dans le cache en temps constant.

Les résultats des calculs précédents sont donc stockés dans un tableau global de longueur (n + 1), où A[i] contient une solution optimale de G_i, G_i désignant les i premiers sommets et les (i – 1) premières arêtes du graphe d'entrée original (et G_0 est le graphe vide). L'algorithme vérifie d'abord si le tableau A contient déjà la solution pertinente S_1 ; sinon, il calcule S_1 récursivement comme avant et met le résultat en cache dans A. De même pour S_2.

Chacun des (n + 1) sous-problèmes n'est désormais résolu à partir de zéro qu'une seule fois. Correctement implémenté, le temps d'exécution passe d'exponentiel à linéaire. Cette forme particulière d'utilisation d'un cache dans un algorithme s'appelle la **mémoïsation**.

III) CALCUL DES VALEURS OPTIMALES : IMPLÉMENTATION TOP-DOWN

Pour l'instant, nous nous concentrons sur le calcul des poids optimaux des sous-problèmes afin de remonter au poids optimum de la solution finale.

Nous verrons plus tard comment identifier également les sommets de l'ensemble indépendant pondéré optimum à partir des valeurs des poids mémorisées (ce qu'on appelle la reconstruction de la solution optimale).

L'implémentation récursive de l'algorithme avec mémoïsation est appelée **top-down**.

Algorithme top-down pour le calcul des poids optimaux

Entrée : Un graphe de type chemin G avec l'ensemble de sommets $\{v_1, v_2, \ldots, v_n\}$ et un poids non négatif w_i pour chaque sommet v_i.

Sortie : Le poids total du meilleur ensemble indépendant pondéré

```
A : = {0: 0, 1: w1}                    # poids des sous-problèmes

rec_poids_MWIS(Gn) :
    Si n == 0 :opt
    |   Retourner A[0]
    Si n == 1 :
    |   Retourner A[1]

    Si A[n] déjà en cache :
    |   Retourner A[n]
    Sinon :
    |   S1 := rec_poids_MWIS(Gn-1)
    |   S2 := rec_poids_MWIS(Gn-2) + wn
    |   A[n] := max {S1, S2}
    |   Retourner A[n]
```

Le schéma de récursion avec le calcul des poids optimaux est représenté sur la figure 6 :

- On descend tout d'abord l'arbre jusqu'à un cas de base.
- Arrivé au cas de base (n = 0 ou n = 1), on attribue la valeur du poids de base : A[0] = 0 ou A[1] = w_1.
- Si le dernier sommet du graphe de l'étape précédente n'a pas été gardé (cas n°1), alors on remonte la valeur sélectionnée (0 ou w_1).
- Si le dernier sommet du graphe de l'étape précédente a été gardé, alors on remonte la valeur sélectionnée (0 ou w_1) + le poids du sommet qui a été gardé.
- On compare les deux poids remontés et on garde celui qui a la valeur maximale (poids optimum) puis on l'enregistre dans la table des valeurs.
- On applique la même méthode tout le long de la remontée de l'arbre.

https://www.informatique-f1.fr/dp/MWIS/

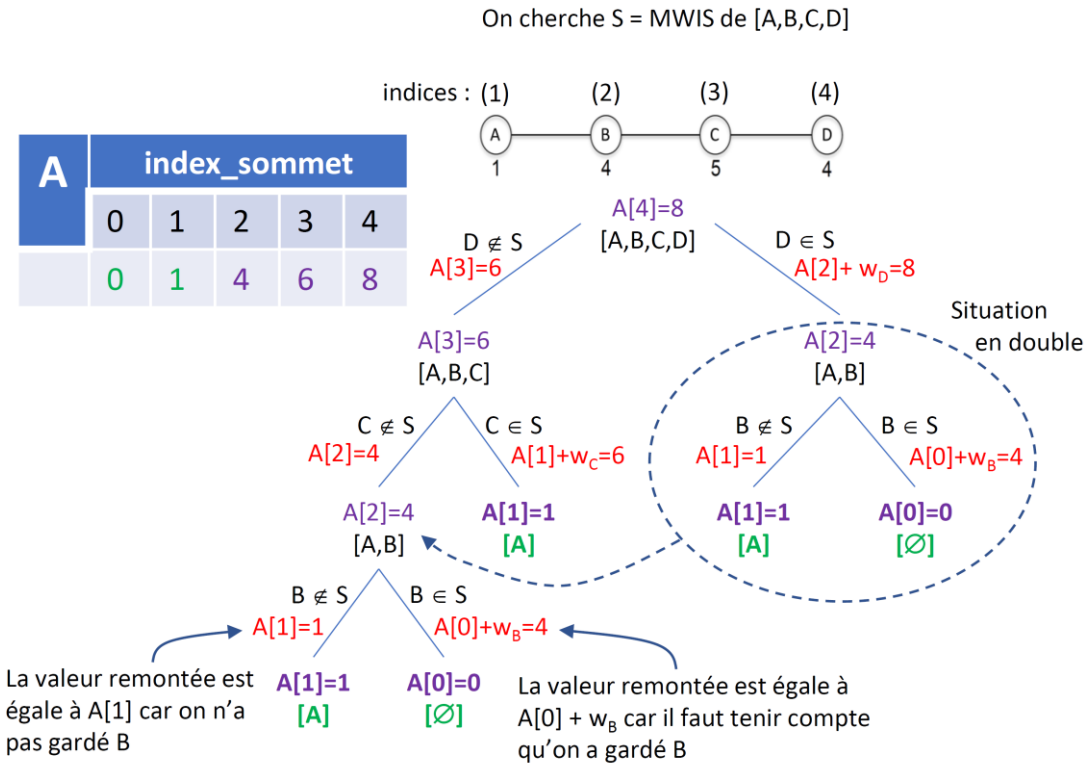


Figure 6 : Arbre de récursion avec calcul des poids optimaux

III.1. Intérêt de la mémoïsation

On remarque que le cas A[2] se présente deux fois, ce qui montre l'importance de la mémoïsation. Dans cet exemple simple, il n'y a qu'un seul cas qui se reproduit, mais on imagine facilement que ce nombre sera beaucoup plus grand dans des situations plus complexes.

La table de mémoïsation nous donne les valeurs optimales des différents sous-problèmes :

- Valeur du MWIS {∅} = A[0] = 0
- Valeur du MWIS {A} = A[1] = 1
- Valeur du MWIS {A, B} = A[2] = 4
- Valeur du MWIS {A, B, C} = A[3] = 6
- Valeur du MWIS {A, B, C, D} = A[4] = 8

Voici une implémentation Python qui retourne les poids optimaux dans un dictionnaire 'A' de l'ensemble des sous-problèmes du graphe :

```python
# graphe à traiter
graphe = {"A":1, "B":4, "C":5, "D":4}

def rec_poids_MWIS(G):
    # Fonction récursive
    def f_rec(Gi, i):
        # Cas de base
        if i < 2:
            return A[i]

        # Récursion sur les autres cas
        if i in A.keys():
            return A[i]
        else:
            # Récupère wi
            wi = list(Gi.values())[i-1]

            # Construction de Gi-1
            Gi = {cle: G[cle] for cle in list(G.keys())[0:i-1]}
            S1 = f_rec(Gi,i-1)

            # Construction de Gi-2
            Gi.popitem()
            S2 = f_rec(Gi,i-2) + wi

            A[i] = max(S1,S2)
            return A[i]

    # Poids optimaux des cas de base
    A = {0:0, 1:graphe["A"]}

    # Appel de la fonction récursive
    f_rec(G,len(G))
    return A

A = rec_poids_MWIS(graphe)
```

On obtient : A = {0: 0, 1: 1, 2: 4, 3: 6, 4: 8}

Afin de montrer l'effet de la mémoïsation, nous allons maintenant appliquer notre algorithme sur un exemple un peu plus complexe, et récupérer le nombre total de récurrences rencontrées et le nombre total de cas qui ont été traités (hors cas de base).

Le graphe traité est le suivant :

Voici le programme Python utilisé :

```python
def rec_poids_MWIS_avec_comptage(G):
    # Fonction récursive
    def f_rec(Gi, i):
        # Comptage du nombre de récurences
        global nbr_cas
        global nbr_cas_hors_base

        nbr_cas = nbr_cas + 1

        # Cas de base
        if i < 2:
            return A[i]

        # Récursion sur les autres cas
        if i in A.keys():
            return A[i]
        else:
            # Incrémente le nombre de calculs hors cas de base
            nbr_cas_hors_base = nbr_cas_hors_base + 1

            # Récupère wi
            wi = list(Gi.values())[i-1]

            # Construction de Gi-1
            Gi = {cle: G[cle] for cle in list(G.keys())[0:i-1]}
            S1 = f_rec(Gi,i-1)

            # Construction de Gi-2
            Gi.popitem()
            S2 = f_rec(Gi,i-2) + wi

            A[i] = max(S1,S2)
            return A[i]

    # Poids optimaux des cas de base
    A = {0:0, 1:graphe["A"]}

    # Nombre de calculs
    global nbr_cas
    global nbr_cas_hors_base
    nbr_cas = 0
    nbr_cas_hors_base = 0

    # Appel de la fonction récursive
    f_rec(G,len(G))
    return A, nbr_cas, nbr_cas_hors_base

# graphe à traiter
graphe = {"A":3, "B":2, "C":1, "D":6, "E":4, "F":5}
A, nbr_cas, nbr_cas_hors_base = rec_poids_MWIS_avec_comptage(graphe)
```

Les valeurs optimales des sous-problèmes obtenues sont les suivantes :

$$A = \{0: 0, 1: 3, 2: 3, 3: 4, 4: 9, 5: 9, 6: 14\}$$

De plus, les résultats montrent que le nombre de récurrences est de 11 et le nombre de cas traités (hors cas de base) est de 5.

Le nombre total de récurrences est donc bien supérieur à $2^{6/2} = 2^3 = 8$ ce qui montre que sans mémoïsation l'algorithme aurait un temps d'exécution exponentiel.

Le nombre de calculs effectués étant en O(n), et chaque calcul prenant un temps en O(1), on a bien un algorithme en O(n).

III.2. Quelques remarques sur les algorithmes top-down

On observe ici que, pour le problème de MWIS sur un graphe de type chemin, l'algorithme top-down avec mémoïsation explore en fait tous les sous-problèmes possibles : il effectue (n − 1) calculs « réels » parmi les (n + 1) sous-problèmes distincts (de G_0 à G_n). Il résout donc exactement (n + 1) sous-problèmes, dont (n − 1) sont non triviaux, jamais moins.

Il existe cependant de nombreux problèmes où l'utilisation d'algorithmes top-down avec mémoïsation permet aussi de réduire drastiquement le nombre de cas réellement calculés par rapport au nombre total de sous-problèmes.

Voici quelques exemples dont certains sont référencés dans votre programme :

- Le problème de *partitionnement équilibré d'entiers positifs*, qui consiste à découper un ensemble d'entiers positifs en deux sous-ensembles dont les sommes sont aussi proches que possible (idéalement égales) ;
- Le problème *du sac à dos*, qui consiste à choisir, parmi des objets ayant chacun un poids et une valeur, un sous-ensemble qui rentre dans un sac de capacité limitée tout en maximisant la valeur totale.
- Le problème de la *distance d'édition de Levenshtein*, qui consiste à mesurer à quel point deux chaînes de caractères sont différentes, en comptant le nombre minimal d'opérations (insertions, suppressions, substitutions) nécessaires pour transformer l'une en l'autre.
- Le problème *d'alignement de séquences* (type Needleman–Wunsch), qui consiste à aligner deux séquences (par exemple d'ADN ou de caractères) en insérant éventuellement des « trous » afin de maximiser une mesure de similarité (ou minimiser un coût de différences).
- Les *algorithmes de plus courts chemins* comme Bellman–Ford et Floyd–Warshall qui calculent les plus courts chemins entre des sommets d'un graphe (Bellman–Ford depuis une source vers tous les sommets, même avec des poids négatifs, et Floyd–Warshall entre tous les couples de sommets).

En conclusion, nous n'avons pas encore exploré tous les avantages des algorithmes top-down !

IV) CALCUL DES VALEURS OPTIMALES : IMPLÉMENTATION BOTTOM-UP

Nous continuons de nous concentrer sur le calcul des poids optimaux des solutions aux sous-problèmes afin de remonter au poids optimum de la solution finale. Cette fois nous allons explorer l'algorithme de type **bottom-up**.

L'idée est ici de résoudre systématiquement l'ensemble des sous-problèmes un par un, mais en partant des cas de base. En effet, la solution d'un sous-problème dépend des solutions de deux sous-problèmes plus petits. Pour s'assurer que ces deux solutions soient immédiatement disponibles, on peut travailler de manière ascendante (bottom-up), en commençant par les cas de base et en construisant progressivement des sous-problèmes de plus en plus grands de manière itérative.

Algorithme bottom-up pour le calcul des poids optimaux

Entrée : Un graphe de type chemin G avec l'ensemble de sommets $\{v_1, v_2, \ldots, v_n\}$ et un poids non négatif w_i pour chaque sommet v_i.

Sortie : Le poids total du meilleur ensemble indépendant pondéré

A : = {0: 0, 1: w_1} # poids des solutions sous-optimales

Pour i allant de 2 à n :

    # Utilisation de l'équation de récurrence
    A[i] := max {A[i – 1], A[i – 2] + w_i}

Retourner A

On pourrait également utiliser un tableau de valeurs de longueur (n + 1) et indexé de 0 à n pour enregistrer les poids optimaux. Au moment où une itération de la boucle principale doit calculer la solution du sous-problème A[i], les valeurs A[i−1] et A[i−2] des deux sous-problèmes plus petits pertinents ont déjà été calculées lors des itérations précédentes (ou dans les cas de base). Ainsi, chaque itération de la boucle prend un temps O(1), pour un temps d'exécution ultra rapide en O(n).

Par exemple, pour le graphe ci-dessous :

… on obtient le tableau suivant :

0	1	2	3	4	5	6
0	3	3	4	9	9	14

À la fin de l'algorithme, chaque case du tableau A[i] contient le poids total d'un MWIS du graphe G_i, qui est composé des i premiers sommets et des (i – 1) premières arêtes du graphe d'entrée. Dans l'exemple ci-dessus, le poids total d'un MWIS du graphe d'entrée original est la valeur de la dernière case du tableau (14), correspondant à l'ensemble indépendant constitué des premier, quatrième et sixième sommet.

Voici un exemple d'implémentation en Python :

```python
def poids_MWIS_bottom_up(G):
    # Poids optimaux des cas de base
    A = {0:0, 1:G["A"]}

    for i in range(2, len(G)+1):
        A[i] = max(A[i-1],A[i-2] + list(G.values())[i-1])

    return A

# graphe à traiter
graphe = {"A":3, "B":2, "C":1, "D":6, "E":4, "F":5}
A = poids_MWIS_bottom_up(graphe)
```

V) ALGORITHME DE RECONSTRUCTION

Les algorithmes que nous avons vus ne calculent que les poids des sous-problèmes optimaux, et non pas le meilleur ensemble indépendant pondéré lui-même.

L'approche pour obtenir le meilleur ensemble indépendant pondéré consiste à utiliser une étape de post-traitement pour le reconstruire à partir des valeurs optimales obtenues par les algorithmes précédents.

Nous avons vu précédemment (voir II.1., aux pages 4 et 5) que deux cas sont possibles pour savoir si un sommet v_n du graphe d'entrée G appartient à l'ensemble indépendant final optimal :

- Si $A[n - 1] \geq A[n - 2] + w_n$, cela signifie que la solution optimale de G_{n-1} est également la solution de notre ensemble final optimum. Dans ce cas, on ne garde pas v_n.
- Sinon, cela signifie que la solution optimale de $G_{n-2} \cup \{v_n\}$ est également une solution optimale de notre ensemble final optimum. Dans ce cas, on garde v_n.

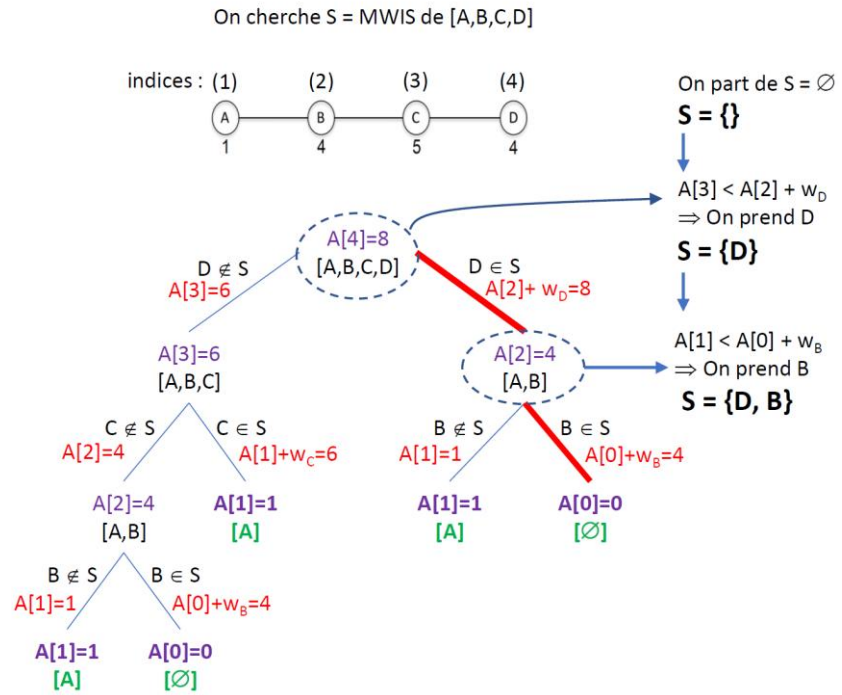


Figure 7 : Principe de reconstruction du graphe pondéré indépendant optimum

L'algorithme de reconstruction est le suivant :

Algorithme de reconstruction de l'ensemble pondéré indépendant optimum

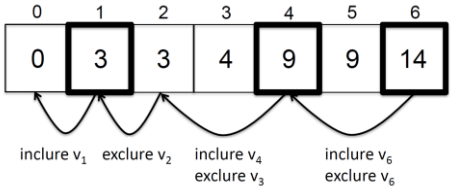
Entrée : le tableau A calculé par l'algorithme top-down ou bottom-up pour un graphe chemin G avec ensemble de sommets {v1, v2, …, vn} et un poids non négatif wi pour chaque sommet vi.
Sortie : un ensemble indépendant de poids maximal de G.

```
S := ∅                                  # Enregistre les sommets de l'ensemble optimum
i := n

Tant que i ≥ 2 :
    Si A[i – 1] ≥ A[i – 2]  + wi :      # Cas n°1
    |    i := i – 1                     # Ne prend pas vi
    Sinon :
    |    S := S ∪ {vi}                  # Cas n°2, on prend vi
    |    i := i – 2                     # On exclut vi-1
Si i == 1 :                             # Cas de base #2
|   S := S ∪ {v1}

Inverser S
Retourner S
```

L'algorithme de reconstruction effectue un seul parcours en arrière sur le tableau A et passe un temps O(1) par itération de boucle, donc il s'exécute en temps O(n).

Par exemple, pour le graphe d'entrée :

… l'algorithme de reconstruction inclut v_6 (ce qui force l'exclusion de v_5), inclut v_4 (ce qui force l'exclusion de v_3), exclut v_2 et inclut v_1 :



Ce qui conduit à la solution optimale {A, D, F}.

Pour un même schéma de programmation dynamique, les versions top-down avec mémoïsation et bottom-up ont en général la même complexité asymptotique en pire cas, car elles résolvent le même ensemble de sous-problèmes.

Toutefois, le top-down mémoisé ne calcule que les sous-problèmes effectivement atteints à partir du problème initial, ce qui fait qu'en pratique, il peut résoudre moins de cas que le bottom-up, qui remplit systématiquement toute la table.

Voici une implémentation en Python :

```
def reconstruction(G,A):
    S = []
    i = len(G)

    while i >= 2:
        if A[i-1] >= A[i-2] + list(G.values())[i-1]:
            i = i-1
        else:
            S.append(list(G.keys())[i-1])
            i = i - 2

    if i == 1:
        S.append(list(G.keys())[0])

    return [S[i] for i in range(len(S)-1,-1,-1)]

# graphe à traiter
graphe = {"A":3, "B":2, "C":1, "D":6, "E":4, "F":5}
A = poids_MWIS_bottom_up(graphe)
S = reconstruction(graphe,A)
```

VI) LES PRINCIPES DE LA PROGRAMMATION DYNAMIQUE

Le concept général de la programmation dynamique peut se résumer en trois étapes. Il se comprend mieux à travers des exemples ; nous n'en avons pour l'instant qu'un seul, mais nous étudierons d'autres cas.

Les trois grands principes de la programmation dynamique

1. Identifier une collection relativement petite de sous-problèmes.
2. Montrer comment résoudre rapidement et correctement les « grands » sous-problèmes à partir des solutions des « plus petits ».
3. Montrer comment déduire rapidement et correctement la solution finale à partir des solutions de tous les sous-problèmes.

Une fois ces trois étapes mises en place, l'algorithme de programmation dynamique peut se mettre en place : on résout tous les sous-problèmes soit :

- un par un, en allant soit du « plus petit » au « plus grand » (méthode de type bottom-up itératif) ;
- soit du « plus grand » au « plus petit » (méthode de type top-down récursive), puis on extrait la solution finale à partir de celles des sous-problèmes.

VI.1. Propriétés souhaitables des sous-problèmes

La clé qui permet de débloquer tout le potentiel de la programmation dynamique pour résoudre un problème, c'est l'identification de la bonne collection de sous-problèmes. En supposant que l'on effectue au moins une quantité de travail constante pour résoudre chaque sous-problème, le nombre de sous-problèmes constitue une borne inférieure sur le temps d'exécution de notre algorithme. Ainsi, on aimerait que ce nombre soit aussi faible que possible. Par exemple, notre solution pour le meilleur chemin indépendant pondéré n'utilisait qu'un nombre linéaire de sous-problèmes, ce qui est généralement le meilleur scénario.

De même, le temps nécessaire pour résoudre un sous-problème (étant données les solutions des plus petits sous-problèmes) et pour déduire la solution finale intervient aussi dans le temps d'exécution global de l'algorithme.

Par exemple, supposons qu'un algorithme résolve au plus f(n) sous-problèmes différents (en les traitant systématiquement du « plus petit » au « plus grand »), en utilisant au plus g(n) de temps pour chacun, et effectue au plus h(n) de travail de post-traitement pour extraire la solution finale (où n désigne la taille de l'entrée). Le temps d'exécution de l'algorithme est alors au plus :

$$\underbrace{f(n)}_{\text{\# sous-problèmes}} \times \underbrace{g(n)}_{\text{temps par sous-problème}} + \underbrace{h(n)}_{\text{post-traitement}}$$

Ces trois étapes demandent de garder respectivement f(n), g(n) et h(n) aussi petits que possible. Dans notre exemple de base, sans l'étape de post-traitement de reconstruction, on a f(n) = O(n), g(n) = O(1) et h(n) = O(1), soit un temps d'exécution global en O(n). Si l'on inclut l'étape de reconstruction, le terme h(n) passe à O(n), mais le temps d'exécution total O(n) × O(1) + O(n) = O(n) reste linéaire.

VI.2. Programmation dynamique vs diviser pour régner

On peut remarquer qu'il y a certaines similitudes entre la méthode « diviser pour régner » et la programmation dynamique, en particulier dans la formulation récursive top-down de cette dernière. Les deux méthodes résolvent récursivement des sous-problèmes plus petits et combinent leurs résultats pour obtenir une solution au problème initial. Voici six différences entre les usages typiques de ces deux méthodes :

1. Chaque appel récursif d'un algorithme typique de type « diviser pour régner » se fixe une seule manière de découper l'entrée en sous-problèmes plus petits. Par exemple, dans l'algorithme de tri par fusion, chaque appel récursif divise son tableau d'entrée en moitié gauche et moitié droite. L'algorithme de tri rapide appelle une procédure de partitionnement pour choisir comment découper le tableau d'entrée en deux, puis se tient à cette division pour le reste de son exécution.
   Chaque appel récursif d'un algorithme de programmation dynamique, lui, garde ses options ouvertes : il considère plusieurs façons de définir des sous-problèmes plus petits et choisit la meilleure. Dans notre exemple d'illustration, chaque appel récursif choisit

entre un sous-problème avec un sommet en moins et un sous-problème avec deux sommets en moins.

2. Comme chaque appel récursif d'un algorithme de programmation dynamique teste plusieurs choix de sous-problèmes plus petits, les mêmes sous-problèmes réapparaissent en général dans différents appels récursifs ; mettre en cache les solutions des sous-problèmes devient alors une optimisation évidente.
   Dans la plupart des algorithmes de type « diviser pour régner », tous les sous-problèmes sont distincts et il n'y a aucun intérêt à mettre leurs solutions en cache. Par exemple, dans les algorithmes de tri par fusion et de tri rapide, chaque sous-problème correspond à un sous-tableau différent du tableau d'entrée.

3. La plupart des applications « classiques » de la méthode « diviser pour régner » consistent à remplacer un algorithme simple en temps polynomial par une version plus rapide. Par exemple, l'algorithme de tri par fusion fait passer le temps d'exécution du tri d'un tableau de $O(n^2)$ à $O(n \log n)$.
   Les meilleurs algorithmes de la programmation dynamique, eux, sont des algorithmes en temps polynomial pour des problèmes d'optimisation dont les solutions naïves (comme la recherche exhaustive) nécessitent un temps exponentiel.

4. Dans un algorithme « diviser pour régner », les sous-problèmes sont choisis principalement pour optimiser le temps d'exécution ; la validité des résultats se vérifie souvent assez facilement. Par exemple, l'algorithme de tri rapide trie toujours correctement le tableau d'entrée, quels que soient la qualité ou le choix de ses éléments pivots.
   En programmation dynamique, on choisit les sous-problèmes pour qu'ils soient mathématiquement suffisants pour reconstruire une solution optimale. Si on se trompe là-dessus, l'algorithme devient faux, même s'il est rapide.

5. Dans un algorithme « diviser pour régner », on applique en général la récursion à des sous-problèmes dont la taille est au plus une fraction constante (par exemple 50 %) de la taille de l'entrée.
   La programmation dynamique, elle, n'a aucun scrupule à appeler la récursion sur des sous-problèmes à peine plus petits que l'entrée, si c'est nécessaire pour la correction.

6. On peut voir la méthode « diviser pour régner » comme un cas particulier de la programmation dynamique, dans lequel chaque appel récursif choisit une collection fixe de sous-problèmes à résoudre récursivement.
   En tant que méthode plus sophistiquée, la programmation dynamique s'applique à une plus grande variété de problèmes que la méthode « diviser pour régner », mais elle est aussi plus exigeante techniquement à utiliser.

Face à un nouveau problème à résoudre, s'il existe une solution évidente de type « diviser pour régner », il faut l'utiliser. Si toutes les tentatives pour trouver une solution « diviser pour régner » échouent, et surtout si elles échouent parce que l'étape de combinaison semble toujours nécessiter de refaire beaucoup de calculs depuis zéro, il est temps d'essayer la programmation dynamique.