Chapitre 7

Techniques algorithmiques

Sommaire

Une grande partie des algorithmes vus dans les chapitres précédents appartiennent à trois grandes familles :

1. les algorithmes *gloutons* (Dijkstra, Prim, Kruskal, ...) ;
2. les algorithmes *"diviser pour régner"* ;
3. les algorithmes de *programmation dynamique* (Bellman-Ford, Floyd-Warshall, ...).

On va examiner plus en profondeur les principes de ces algorithmes dans ce chapitre. Les techniques que l'on y couvrira s'appliquent aux problèmes :

— de décision, où l'on doit répondre "oui" ou "non" à une question ;
— de recherche, où l'on doit trouver un / plusieurs / tous les élément(s) satisfaisant un certain critère ;
— de tri, où l'on doit ordonner une structure selon un critère particulier ;
— d'optimisation, où l'on doit renvoyer une solution de coût est optimal, c'est-à-dire le plus petit ou le plus grand possible selon qu'on s'intéresse à un problème de minimisation ou de maximisation.

7.1 Sous-problèmes et sous-solutions optimales

La notion de *sous-problème* interviendra dans les trois paradigmes que nous examinerons dans ce chapitre : qu'il s'agisse d'algorithmes gloutons, de "diviser pour régner" ou de programmation dynamique, il est essentiel pour que ces algorithmes fonctionnent que le problème à résoudre se factorise en des sous-problèmes analogues sur des données réduites, et que les solutions optimales à ces problèmes soient exploitables dans la suite de l'algorithme. Les différences entre ces techniques résident principalement dans la manière de traiter et d'exploiter certains de ces sous-problèmes.

Cette notion de sous-problème est intuitive et a déjà été rencontrée dans d'autres cours, notamment dans le cadre de la récursivité : il s'agit simplement de résoudre le problème de départ, mais sur des instances plus petites en laquelle l'instance originale se décompose. Par exemple :

— la recherche dichotomique résout le problème de recherche d'un élément dans un tableau en le recherchant dans des sous-tableaux,
— les algorithmes de tri QuickSort et MergeSort trient tous deux un tableau en triant certains de ses sous-tableaux.

7.2 Algorithmes gloutons

7.2.1 Principe

De nombreux algorithmes construisent leur solution à un problème donné pas à pas (itérativement ou récursivement). Les algorithmes gloutons construisent cette solution en rajoutant à chaque étape le candidat optimal à la solution partielle en cours de construction (ce qu'on appelle parfois un *choix glouton*). L'argument utilisé pour justifier l'utilisation de cette approche affirme que faire le "meilleur" choix à chaque étape implique que le résultat basé sur tous ces meilleurs choix est forcément aussi le meilleur.

Les algorithmes gloutons semblent intuitivement corrects et pleins de bon sens. En réalité, l'intuition peut se révéler fausse, et rien ne garantit *a priori* qu'effectuer des choix *localement* optimaux mène à une solution *globalement* optimale. Les cas où ils fournissent une solution optimale sont d'ailleurs relativement rares [1] ; il s'agit néanmoins d'une méthode qu'il est important de connaître et de pouvoir appliquer, car elle est simple à appréhender et permet au moins d'avoir un point de départ pour tenter de résoudre un problème. De nombreux cas existent où la méthode gloutonne ne fournit pas une solution optimale, mais où l'on constate qu'elle se comporte de manière satisfaisante en pratique. Il arrive également souvent que l'on puisse prouver des *garanties d'approximation*, c'est-à-dire que l'on puisse garantir par exemple qu'un algorithme non optimal donnera au pire une solution dont la valeur est 1.5, 2 ou 3 fois celle de la solution optimale (pour un problème de minimisation).

Un argument en défaveur des algorithmes gloutons, par rapport à d'autres techniques qu'on verra plus loin, est que les choix qu'ils font ne dépendent que de la situation actuelle dans l'algorithme sans vision globale sur l'ensemble du problème. De plus, une fois qu'un choix est fixé, on ne revient pas (ou peu) sur les choix précédents, ce qui nous limite grandement et peut nous mener à accumuler beaucoup d'erreurs.

1. Une théorie mathématique basée sur les *matroïdes* caractérise précisément les cas où ces algorithmes fonctionnent. Le lecteur curieux consultera Cormen et al. [1].

7.2.2 Exemples déjà rencontrés

On a déjà rencontré quelques exemples d'algorithmes gloutons dans les chapitres précédents. Réexaminons-les brièvement.

Forêts couvrantes de poids minimum

Les deux algorithmes de recherche d'un arbre (ou d'une forêt) couvrant(e) de poids minimum que l'on a vus, à savoir celui de Prim et celui de Kruskal, sont tous deux des algorithmes gloutons. En effet, les deux algorithmes partent d'une structure initialement vide, à laquelle on rajoute à chaque étape une arête sûre[2] jusqu'à ce que ce ne soit plus possible. Il s'agit donc bien d'une stratégie gloutonne : pour construire une structure de poids minimum, on ajoute à chaque étape une arête de poids minimum satisfaisant nos critères. La seule différence entre les deux algorithmes est que celui de Prim se force à maintenir un arbre à chaque itération, et nous oblige donc à choisir un sommet de départ par composante connexe, alors que celui de Kruskal s'affranchit de cette contrainte.

L'algorithme de Dijkstra

L'algorithme de Dijkstra est également un algorithme glouton ; pour calculer les distances vers tous les sommets à partir d'un sommet source :

1. on part d'un ensemble de distances calculées et d'un ensemble S de sommets déjà traités ;
2. à chaque étape, on sélectionne le sommet le plus proche de l'ensemble des sommets déjà traités pour découvrir de nouveaux chemins ou améliorer les chemins déjà connus.

Le critère de sélection ici est la distance du nouveau sommet à l'ensemble déjà traité, qui intuitivement semble bien correspondre à ce que l'on veut puisque l'on cherche à construire tous les plus courts chemins vers les autres sommets du graphe.

7.2.3 Stratégie de preuve de correction

Pour prouver qu'un algorithme glouton est correct (ce qui n'est, encore une fois, pas toujours vrai), une stratégie qui a fait ses preuves est la suivante :

1. on suppose qu'il existe une solution optimale T qui est différente de la solution gloutonne S ;
2. on cherche la "première" différence entre ces deux solutions ;
3. on montre que l'on peut remplacer la décision prise à cette étape par un choix glouton sans dégrader la qualité de T.

En réitérant cette transformation autant de fois que nécessaire, on en conclut que toute solution optimale peut être transformée en une solution gloutonne sans dégrader sa qualité, et donc que la stratégie gloutonne est optimale.

2. Rappelons qu'une arête sûre est une arête valide (donc qui ne crée pas de cycle dans la structure en cours de construction) de poids minimum.

On a vu une illustration de ce principe dans le cas de l'algorithme de Prim, où l'on a montré par induction qu'il existait toujours une solution optimale (globale) qui contenait notre sous-solution gloutonne (partielle). Notons qu'il est parfois nécessaire (on verra si on en a besoin) de montrer qu'aucune autre solution optimale ne fait mieux que l'approche gloutonne.

Prouvons que ces deux algorithmes sont corrects en suivant la stratégie exposée ci-dessus. On supposera que Prim est relancé pour chaque composante connexe du graphe.

Proposition 7.2.1. Les algorithmes de Prim et de Kruskal fournissent bien des forêts couvrantes de poids minimum.

Démonstration. Soit $S = (e_1, e_2, \ldots, e_n)$ la liste d'arêtes sélectionnées dans cet ordre par l'algorithme de Prim (ou de Kruskal), et $T = (f_1, f_2, \ldots, f_n)$ une solution optimale, qui a forcément le même nombre d'arêtes. Soit i l'indice de la première différence entre S et T : on a donc $T = (e_1, e_2, \ldots, e_{i-1}, f_i, f_{i+1}, \ldots, f_n)$.

Remplaçons maintenant l'arête f_i par l'arête e_i, et examinons la solution résultante T' : son coût est donné par $w(T') = w(T) - w(f_i) + w(e_i)$, et comme $w(f_i) \leq w(e_i)$ puisque e_i est sûre, on a $w(T') \leq w(T)$. Il nous suffit de réitérer ce processus jusqu'à ce que $T = S$, et comme les transformations ne dégradent jamais la qualité de la solution optimale, le résultat S est bien optimal. □

Il n'est pas nécessaire, dans la preuve ci-dessus, de montrer que la transformation appliquée à chaque étape conserve la structure de forêt couvrante, car on pourrait opposer l'argument que l'on applique les transformations jusqu'à obtenir S, dans laquelle on a bien veillé à obtenir une forêt couvrante.

7.2.4 Cas où la stratégie gloutonne n'est pas optimale

Afin d'éviter de penser que la stratégie gloutonne fonctionne toujours, nous allons examiner quelques exemples de problèmes où elle semble intuitivement correcte mais où elle ne fournit pas une solution optimale. Cela arrive fréquemment lorsqu'on se confronte à des problèmes dits "NP-complets", pour lesquels il est peu probable qu'un algorithme "efficace" existe.

Ces notions seront précisées dans des cours ultérieurs plus avancés ; voici déjà une présentation informelle qui nous suffira dans ce cours. On dit d'un problème qu'il est :

- *dans (la classe)* P s'il existe un algorithme de complexité polynomiale en la taille de l'instance qui résout ce problème ;
- *dans (la classe)* NP s'il existe un algorithme de complexité polynomiale en la taille de l'instance qui *vérifie* une solution à ce problème ;
- NP-*complet* si tout problème de la classe NP peut se ramener à ce problème par un mécanisme de "réduction".

Être capable de vérifier une solution en temps polynomial n'implique pas forcément qu'on soit capable de la trouver. Cette différence est centrale à une question importante d'informatique théorique : "est-ce que P=NP ?". Le consensus actuel est que les deux classes sont bel et bien différentes, et qu'il existe donc des problèmes dans NP qu'il est impossible de résoudre en temps polynomial ; mais aucune preuve de cette affirmation n'existe à ce jour.

La leçon qu'on en tire en pratique est la suivante : si le problème qu'on doit résoudre est NP-complet, il est fort peu probable qu'un algorithme polynomial puisse le résoudre, et encore moins si cet algorithme est glouton.

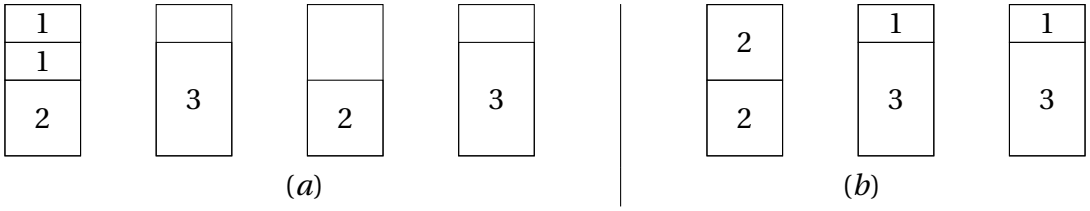
Bin packing

Le problème connu sous le nom de *bin packing* apparaît naturellement dans un grand nombre d'applications. Il s'agit, étant donné un ensemble de boîtes d'une capacité B et un ensemble S d'objets dont on connaît les tailles, de ranger ces objets dans les boîtes de manière à utiliser le moins de boîtes possibles. Plus formellement :

BIN PACKING
Entrées: un ensemble S d'objets, une taille $t(s)$ naturelle $\forall s \in S$, une capacité naturelle B et un naturel K.
Question: existe-t-il une partition $S = S_1 \cup S_2 \cup \cdots \cup S_K$ telle que $\sum_{x \in S_i} s(x) \leq B \ \forall 1 \leq i \leq K$?

Ce problème est NP-complet [2], et l'on ne s'attend donc pas à ce qu'il existe un algorithme polynomial pouvant le résoudre de manière efficace. En revanche, plusieurs stratégies gloutonnes ont été proposées pour le résoudre : une de ces stratégies, nommée *first-fit*, consiste à sélectionner comme boîte candidate la première parmi celles déjà utilisées dans laquelle il reste suffisamment d'espace pour accueillir l'objet sélectionné (si l'on n'en trouve pas, on rajoute une nouvelle boîte). Cette stratégie n'est pas optimale, mais on peut prouver que la qualité de la solution produite est au pire 2 fois celle de la solution optimale — autrement dit, dans le pire des cas, on utilise deux fois trop de boîtes [3]. On peut parfois améliorer la qualité de la solution en commençant par les objets les plus gros, mais il a été prouvé qu'il n'existe pas d'algorithme polynomial permettant d'obtenir une meilleure garantie que 1.5 [4].

Exemple 52. Voici un cas où la stratégie "first fit" n'est pas optimale. On reçoit des objets de tailles 2, 3, 1, 2, 3, 1 que l'on examine dans cet ordre pour les ranger au fur et à mesure dans des boîtes de capacité 4. (*a*) La stratégie "first fit" fournit une solution utilisant 4 boîtes ; (*b*) on peut faire mieux en n'utilisant que trois boîtes, ce qui est bien optimal puisque la somme des poids (12) divisée par la capacité des boîtes (4) nous oblige à utiliser au moins 3 boîtes :



On peut facilement vérifier que "first fit" ne se débrouille pas mieux si l'on trie les objets par poids, que ce soit de manière croissante ou décroissante.

L'algorithme 37 implémente la stratégie gloutonne que l'on vient d'exposer.

Minimum common superstring (ou supersequence)

De nombreuses applications requièrent de chercher une ou plusieurs sous-chaînes dans une grande chaîne. On peut également prendre le problème dans l'autre sens et se deman-

3. Ceci est une estimation pessimiste : en réalité, le taux est de 1.7 — voir Dósa et Sgall [3].

Algorithme 37 : BINPACKINGFIRSTFIT(S, K)

Entrées : une liste S de poids d'objets, une capacité K.
Sortie : une partition de S sous la forme d'une liste de listes, où la somme des tailles de chaque part n'excède pas K.

```
1  boîtes ← liste();
2  boîtes.ajouter_en_fin(liste());
3  tailles ← liste();
4  tailles.ajouter_en_fin(0);
5  n ← 0;
6  pour chaque poids ∈ S faire
       // placer l'objet dans la première boîte pouvant l'accueillir
7      trouvé ← FAUX;
8      pour chaque i allant de 0 à n faire
9          si tailles[i] + poids ≤ K alors
10             boîtes[i].ajouter_en_fin(poids);
11             tailles[i] ← tailles[i] + poids;
12             trouvé ← VRAI;
13             arrêter;
       // s'il n'y en a pas, placer l'objet dans une nouvelle boîte
14     si ¬ trouvé alors
15         boîtes.ajouter_en_fin(liste());
16         n ← n + 1;
17         boîtes[n].ajouter_en_fin(poids);
18         tailles.ajouter_en_fin(poids);
19 renvoyer boîtes;
```

der comment représenter un ensemble de chaînes données par une “sur-chaîne” ou une “sur-séquence”, qui est une chaîne contenant toutes les chaînes données en entrée. Rappelons que, étant données deux chaînes A de longueur k et B de longueur n :

— A est une *sous-chaîne* de B, et B est une *sur-chaînes* de A, s'il existe un indice i tel que $B_iB_{i+1}\dots B_{i+k-1} = A$ (autrement dit, les caractères de l'occurence de A dans B doivent être consécutifs) ;

— A est une *sous-séquence* de B, et B est une *sur-séquence* de A, s'il existe k indices $i_1 < i_2 < \dots < i_k$ tels que $B_{i_1}B_{i_2}\dots B_{i_k} = A$ (autrement dit, les caractères de l'occurence de A dans B ne doivent pas nécessairement être consécutifs) ;

Ces notions donnent naturellement lieu à l'étude du problème suivant, dont on retrouve des applications en bioinformatique, plus précisément en ce qui concerne la reconstruction d'un génome.

PLUS COURTE SUR-SÉQUENCE
Entrées: un ensemble S de chaînes de caractères, une taille K.
Question: existe-t-il une chaîne C de taille K telle que toute chaîne de S est une sous-chaîne de C ?

Une simple concaténation des éléments de S nous donne une solution à ce problème, mais rien ne garantit qu'elle est la plus courte possible. Une stratégie gloutonne qui a été proposée est la suivante : à chaque étape, on repère dans l'ensemble S deux chaînes A et B dont le *chevauchement*, c'est-à-dire la taille du plus long suffixe de A qui est aussi un préfixe de B (ou le contraire), est maximal, et on remplace dans S ces deux chaînes par leur plus courte sur-chaîne. L'algorithme s'arrête quand il n'y a plus de chaîne à fusionner — autrement dit, quand l'ensemble S ne possède plus qu'un élément. L'intuition de cette approche est que plus le chevauchement entre deux chaînes est grand, moins l'on doit rajouter de caractères pour construire une sur-chaîne les contenant toutes les deux.

L'algorithme 38 implémente cette stratégie gloutonne.

Algorithme 38 : PLUSCOURTESURSÉQUENCEGLOUTONNE(S)

Entrées : un ensemble S de chaînes de caractères.
Sortie : une sur-chaîne des éléments de S.

```
1 tant que |S| > 1 faire
2 |   C1, C2 ← deux chaînes de S de chevauchement maximal;
3 |   S ← S \ {C1, C2} ∪ plus courte sur-chaîne de C1 et C2;
4 renvoyer l'unique élément de S;
```

On peut prouver que cette stratégie garantit un taux d'approximation de 2.75, c'est-à-dire que cette heuristique ne renverra jamais de solution plus longue que 2.75 fois la taille de la solution optimale. Voici un exemple de cas où elle n'est pas optimale.

Exemple 53. [5] L'ensemble de chaînes $\{(ab)^kc, (ba)^k, c(ab)^k\}$ admet la solution optimale $c(ab)^{k+1}c$ de taille $2k+4$. Cependant, l'approche gloutonne construira la chaîne $c(ab)^kc(ba)^k$, de longueur $4k+2$ et qui ne sera pas optimale pour $k > 1$.

7.3 Diviser pour régner

L'idée de l'approche "diviser pour régner" consiste à découper un problème donné en sous-problèmes plus petits, puis à recombiner les sous-solutions pour obtenir la solution au problème général. On distingue donc trois étapes :

1. **diviser** : découper le problème en sous-problèmes ;
2. **régner** : résoudre les sous-problèmes récursivement (ou directement s'ils sont de taille suffisamment petite) ;
3. **combiner** : combiner les solutions des sous-problèmes pour obtenir la solution au problème de départ.

Bien entendu, l'efficacité de cette approche dépend fortement des complexités de ces trois phases. On obtient généralement un algorithme efficace quand :

1. le découpage est bien choisi, et
2. combiner deux sous-solutions prend moins de temps que de calculer la solution sans procéder à la division.

La première condition semble évidente à satisfaire, mais on ne maîtrise pas toujours la manière dont les découpes sont réalisées. Un exemple connu de cas où ceci peut être problématique est le choix du pivot dans l'algorithme du tri rapide (*quicksort*) : l'algorithme est en $O(n \log n)$ si "tout se passe bien", mais peut dégénérer en $O(n^2)$ sur certaines entrées si le choix du pivot est mal réalisé.

7.3.1 Dichotomie

La recherche dichotomique est un des premiers exemples d'algorithme efficace suivant la stratégie "diviser pour régner" que l'on couvre dans les cours d'algorithmique. Si la séquence dans laquelle on veut trouver un élément est triée, on peut appliquer la stratégie suivante :

1. **diviser** : on divise l'intervalle de recherche en deux parties égales (à un élément près) ;
2. **régner** : on effectue la recherche récursivement dans une seule des deux sous-parties en fonction de la valeur de l'élément recherché et du pivot sélectionné ;
3. **combiner** : le résultat est la combinaison du résultat de la recherche dans les deux parties.

Ici, le nombre de découpes est en $O(\log n)$, alors que l'étape de combinaison est en $O(1)$: il suffit de renvoyer la position de l'élément dans la partie où la recherche s'est déroulée.

7.3.2 Tri fusion

Un autre exemple d'algorithme efficace suivant la stratégie "diviser pour régner" est le tri fusion, qui trie une séquence de la façon suivante :

1. **diviser** : on divise la séquence en deux parties égales (à un élément près) ;
2. **régner** : on trie récursivement les deux sous-parties séparément ;

3. **combiner** : on fusionne ensuite les deux sous-parties triées de manière à ce que le résultat soit encore trié.

L'analyse de complexité peut s'effectuer en évaluant le nombre de découpages effectués et le temps total mis pour les fusions :

1. la profondeur de l'arbre d'appels récursifs est $O(\log n)$;
2. la "fusion triée" de deux listes triées totalisant ensemble n éléments peut s'effectuer en $O(n)$.

On obtient alors un algorithme de tri en $O(n \log n)$ (si les comparaisons sont en $O(1)$), un gain non négligeable par rapport aux algorithmes naïfs (bulle, sélection et insertion) en $O(n^2)$.

7.3.3 Exponentiation rapide

L'exponentiation rapide est encore un autre exemple d'algorithme suivant la stratégie "diviser pour régner". Au lieu de calculer la puissance x^n naïvement (multiplier x par x jusqu'à obtenir la puissance désirée, ce qui est en $O(n)$), on peut utiliser une manipulation simple pour en déduire un algorithme plus efficace ; on réécrit simplement la puissance comme suit :

$$x^n = \begin{cases} 1 & \text{si } n = 0, \\ (x^2)^{n/2} & \text{si } n > 0 \text{ et pair}, \\ x(x^2)^{(n-1)/2} & \text{si } n \text{ est impair}. \end{cases}$$

Avec une analyse similaire à celle de la complexité de la recherche dichotomique, on en déduit que l'exponentiation rapide se fait en $O(\log n)$, un gain non négligeable par rapport à l'algorithme naïf en $O(n)$.

7.3.4 Autres

L'algorithme de Strassen pour le produit matriciel est aussi un exemple d'algorithme de type "diviser pour régner" plus efficace que l'algorithme naïf permettant de multiplier deux matrices $n \times n$ en temps $O(n^3)$. De manière assez surprenante, l'algorithme de Strassen résout ce problème en temps $O(n^{2,81})$, en partitionnant les matrices en sous-matrices suffisamment petites pour que leur produit se calcule plus rapidement. Le lecteur curieux consultera Cormen et al. [1] pour les détails.

7.4 Programmation dynamique

La **programmation dynamique**[4] est une technique algorithmique que l'on applique en général à des problèmes d'optimisation. La stratégie qu'elle utilise consiste à se baser sur une notion de sous-structure optimale, et à utiliser des solutions optimales à des sous-problèmes que l'on calcule au préalable pour obtenir une solution globale optimale. Les algorithmes se basant sur la programmation dynamique, et les situations dans lesquelles on

4. On doit son invention à Richard Bellman ; pour l'origine du nom, voir Dreyfus [6].

y a recours, s'expliquent mieux en prenant comme point de départ des algorithmes récursifs.

7.4.1 Nombres de Fibonacci

Un exemple simple mais spectaculaire de l'efficacité de la programmation dynamique est son application au calcul des nombres de Fibonacci. Pour rappel, le $n^{\text{ème}}$ nombre de Fibonacci F_n est défini pour toute valeur $n \in \mathbb{N}$ par :

$$F_n = \begin{cases} n & \text{si } n \leq 1, \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

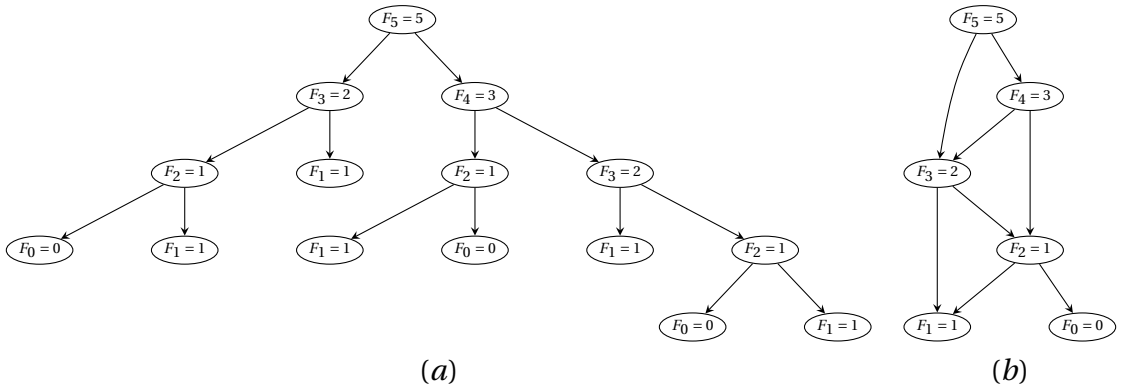
En déduire un algorithme récursif calculant ces nombres est donc un exercice trivial (voir l'algorithme 39). On se rend également compte, en dessinant l'arbre d'appels (voir le cas $n = 5$ à la Figure 7.1(a)), de deux problèmes qui sont liés :

1. cet algorithme est extrêmement inefficace : le calcul de F_n nécessite deux appels récursifs, qui eux-mêmes en nécessitent chacun deux, et ainsi de suite jusqu'à ce qu'on retombe sur les cas de base, ce qui nous donne du $O(2^n)$;
2. on effectue un grand nombre de fois les mêmes appels pour recalculer des valeurs que l'on connaît déjà.

Algorithme 39 : FIBONACCINAÏF(n)

Entrées : un naturel n.
Sortie : le n-ème nombre de Fibonacci.

```
1 si n ≤ 1 alors renvoyer n;
2 renvoyer FIBONACCINAÏF(n − 1) + FIBONACCINAÏF(n − 2);
```



(a) (b)

FIGURE 7.1 – (a) L'arbre d'appels du calcul naïf de F_5 ; (b) le DAG obtenu en fusionnant les sommets du graphe précédent qui correspondent aux mêmes appels.

Lorsqu'on simplifie l'arbre d'appels en fusionnant les sommets correspondant aux appels renvoyant le même résultat, on obtient un DAG bien plus compact (voir la Figure 7.1(b)) qui suggère que l'on doit pouvoir s'en tirer de manière bien plus efficace. C'est bien le cas : au lieu d'effectuer des appels pour recalculer tout le temps les mêmes valeurs, on pourrait

Algorithme 40 : FIBONACCISTOCKAGE(n)

Entrées : un naturel n.
Sortie : le n-ème nombre de Fibonacci.

```
1 si n ≤ 1 alors renvoyer n;
2 valeurs ← tableau(n + 1, 0);
3 valeurs[0] ← 0 ; valeurs[1] ← 1;
4 pour chaque i allant de 2 à n faire
5 |   valeurs[i] ← valeurs[i − 1] + valeurs[i − 2];
6 renvoyer valeurs[n];
```

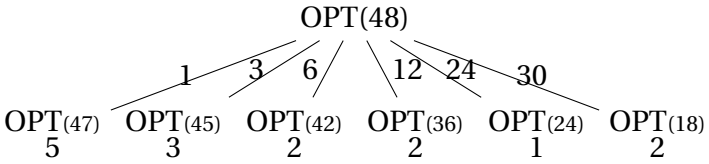
stocker les valeurs déjà calculées dans une table qu'on consulterait à chaque fois que l'on en a besoin. L'algorithme 40 implémente cette approche.

Cette nouvelle version nécessite un espace en $O(n)$ pour stocker `valeurs`, et s'exécute en temps $O(n)$, ce qui correspond au temps nécessaire pour calculer et stocker toutes les valeurs nécessaires au calcul de F_n. Remarquons qu'on peut faire encore mieux, car on n'a en fait pas besoin de stocker toutes les valeurs : seules les deux dernières valeurs sont nécessaires au calcul du nombre qui nous intéressent ; toutes les valeurs précédentes ne servent qu'à calculer ces deux nombres-là, et l'on peut donc ne stocker à chaque étape que les nombres indispensables au calcul des nombres suivants.

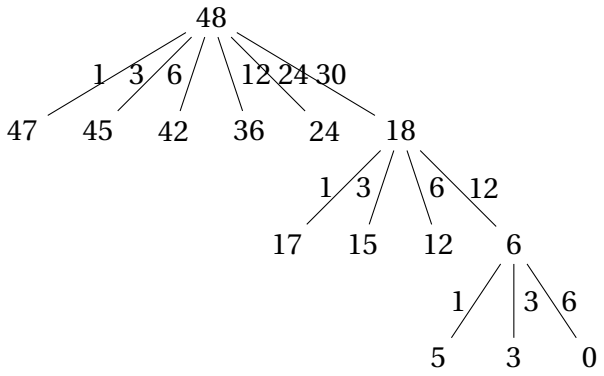
7.4.2 Différences avec l'approche gloutonne

La programmation dynamique peut donner l'impression de ressembler à l'approche gloutonne, mais les deux techniques sont très différentes. Elles examinent toutes deux plusieurs choix à chaque étape, mais l'approche gloutonne se bornera à sélectionner le meilleur choix *local* pour la situation actuelle sans tenir compte du coût global de la solution. Un algorithme de programmation dynamique, au contraire, examinera chaque choix possible à chaque étape sans se débarrasser des choix localement non optimaux, et optimisera la solution renvoyée en prenant en compte à la fois le choix réalisé et les implications pour le sous-problème résiduel, ce qui lui permettra de prendre une meilleure décision.

Exemple 54. Soit $S = \{1, 3, 6, 12, 24, 30, 60, 240\}$, et $M = 48$. Notre but est de trouver le plus petit nombre d'éléments de S, que l'on peut sélectionner chacun autant de fois que l'on veut, dont la somme vaut M. L'approche basée sur la programmation dynamique effectuera les appels suivants (on omet le détail du calcul de chaque sous-solution optimale) :



Sur base de ces informations, on en déduit que la solution optimale de taille 2 s'obtient en effectuant le meilleur choix *global* commençant par sélectionner l'élément 24. L'algorithme glouton, en revanche, procèdera comme suit :



À chaque étape, l'algorithme glouton effectue le meilleur choix *local* qui consiste à réduire au maximum le nombre à décomposer. Malheureusement, ce choix n'est pas globalement optimal, et la solution trouvée (de taille 3) est donc de moins bonne qualité que ce que fournit la programmation dynamique.

(fin exemple 54)

7.4.3 Conception d'algorithmes de programmation dynamique

L'élaboration d'algorithmes de programmation dynamique requiert de mettre en évidence une structure de sous-problèmes dont on peut exploiter les solutions optimales, de manière à pouvoir en déduire une première ébauche récursive dont on supprime ensuite les appels récursifs par des lectures et écritures dans une table de résultats intermédiaires. On applique donc en général la stratégie suivante :

1. caractériser la structure d'une solution optimale ;
2. définir la valeur d'une solution optimale récursivement, ce qui donne lieu à l'écriture d'une *équation de programmation dynamique* ;
3. écrire l'algorithme correspondant sous forme récursive, et enfin
4. remplacer les appels récursifs par un stockage ou une consultation des valeurs calculées ; bien souvent, on se rend ensuite compte qu'il est possible d'optimiser la consommation en mémoire.

Lorsqu'on a l'habitude de manipuler ce type d'algorithmes, on saute en général l'étape d'écriture de l'algorithme sous forme récursive. On conservera cette étape ici pour des raisons pédagogiques.

7.4.4 Un problème de découpe

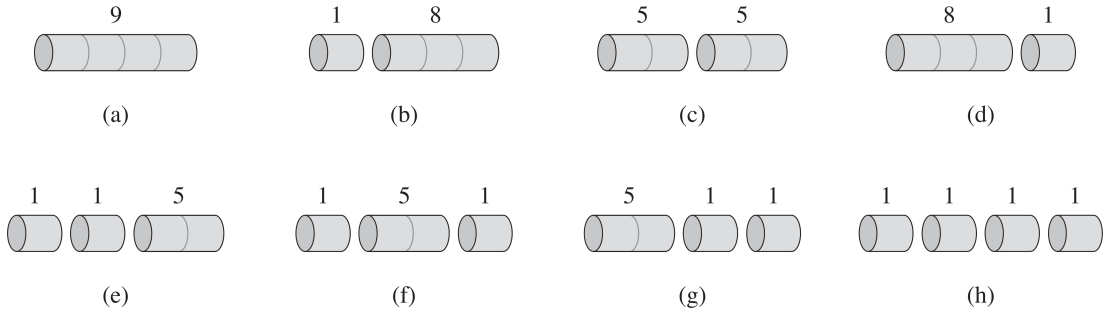
Afin d'examiner les avantages de la programmation dynamique dans le cadre d'un problème d'optimisation, examinons le problème de découpe suivant tiré de Cormen et al. [1]. Une entreprise achète des tiges d'acier, les découpe en morceaux de diverses tailles, et revend ensuite les morceaux. Le prix de chaque morceau dépend de sa taille, et l'entreprise souhaite réaliser la découpe qui lui permettra de maximiser son profit. Plus formellement :

DÉCOUPE
Entrées: une longueur de tige ℓ, un tableau de prix p donnant la valeur de chaque morceau de taille $1, 2, \ldots, \ell$

Objectif: une découpe de la tige en morceaux de longueur $\ell_1, \ell_2, \ldots, \ell_k$ qui maximise $\sum_{i=1}^{k} p[\ell_i]$.

L'exemple 55 permet de fixer les idées et montre toutes les manières de découper une tige de longueur 4 en morceaux entiers. En examinant toutes ces options, on se rend compte que la découpe la plus rentable coupe la tige en deux morceaux de même taille.

Exemple 55. Les 8 manières de découper une tige de longueur 4 en morceaux entiers. Les prix correspondant à chaque morceau sont indiqués au-dessus de ces morceaux, et l'on remarque que la stratégie optimale consiste ici à couper la tige en deux parts égales [1].



Sous-structure optimale et première version récursive

Une manière d'obtenir la structure d'une solution optimale à ce problème consiste à examiner l'impact qu'aurait chaque choix de longueur possible en effectuant une seule découpe ; une fois cette découpe effectuée, on ne touche plus à l'un des deux morceaux, et l'on cherche une découpe optimale de l'autre morceau. Si notre tige est de longueur n, on peut décider de la couper en :

— un morceau de longueur 1 et un morceau de longueur $n-1$, ou
— un morceau de longueur 2 et un morceau de longueur $n-2$, ou
⋮
— un morceau de longeur $n-1$ et un morceau de longueur 1, ou encore
— un morceau de longueur n et rien d'autre (on ne coupe donc pas).

Notons p_i le prix d'un morceau de longueur i. Si une seule coupe est permise, on cherche naturellement celle qui maximisera le prix obtenu, qui est égal à la somme des prix des deux morceaux constituant le découpage. Autrement dit, on cherche la valeur i qui maximise la quantité $p_i + p_{n-i}$.

Mais dans ce problème, un nombre arbitraire de découpes est permis ; il nous faut donc également vérifier si l'un des deux morceaux que l'on a obtenus ne peut pas lui-même être découpé de manière à augmenter le profit. Il suffit d'examiner un seul de ces morceaux, car on suppose que dans la solution que l'on construit, il y aura au moins un morceau de la taille que l'on a sélectionnée.

On va donc décider, une fois qu'on a effectué une découpe, d'examiner les manières optimales de découper l'un des deux morceaux… et ainsi de suite récursivement. Plus formellement, notons r_i le profit que nous donne une découpe optimale d'un morceau de longueur i. La quantité que l'on cherche à obtenir est donc fournie par l'équation de programmation

dynamique suivante :

$$r_n = \begin{cases} p_n & \text{si } n \leq 1, \\ \max_{1 \leq i \leq n}(p_i + r_{n-i}) & \text{sinon.} \end{cases} \tag{7.1}$$

On en déduit donc facilement l'algorithme 41, qui calcule récursivement la *valeur* d'une solution optimale à notre problème. La condition d'arrêt est le cas où la tige fournie est trop courte pour subir une découpe (c'est-à-dire qu'elle est de longueur 0 ou 1).

Algorithme 41 : DÉCOUPEOPTIMALENAÏVE(longueur, prix)

Entrées : une longueur naturelle de tige, un tableau de prix donnant pour chaque longueur possible le prix correspondant.
Sortie : le profit maximal réalisable.

```
   /* aucune découpe possible: renvoyer directement le prix          */
1  si longueur ≤ 1 alors renvoyer prix[longueur];
   /* tester toutes les découpes possibles et garder la meilleure    */
2  meilleur_prix ← prix[longueur];
3  pour chaque i allant de 1 à longueur −1 faire
4  |  meilleur_prix ← max(meilleur_prix, prix[i] +
   |   DÉCOUPEOPTIMALENAÏVE(longueur − i , prix));
5  renvoyer meilleur_prix;
```

Mémoïsation et amélioration de la complexité

L'algorithme 41 fonctionne correctement et nous donnera donc bien la valeur optimale. Malheureusement, il souffre de deux défauts qui sont liés, et qu'on observait déjà dans le calcul des nombres de Fibonacci :

1. sa complexité est inacceptable : si n est la longueur donnée, on se retrouve dans chaque appel de fonction à effectuer $O(n)$ appels récursifs, et la profondeur de l'arbre d'appels est également en $O(n)$ puisqu'une découpe ne diminue dans le pire des cas la taille du morceau que de 1 ; on a donc une complexité — grossièrement estimée — de $O(n^n)$;
2. de nombreux appels récursifs sont inutiles, car ils nous servent à calculer des valeurs que l'on connaît déjà.

On va améliorer les choses en stockant les solutions optimales des sous-problèmes. On effectuera ce stockage de manière croissante sur les longueurs de tige possibles, ce qui nous permettra grâce à l'Équation 7.1 de récupérer les valeurs des sous-découpes optimales pour toutes les longueurs inférieures à celle qu'on est en train d'examiner. Ce procédé s'appelle la *mémoïsation*[5]. On remplira donc une table de profits comme suit :

— on ne peut pas couper une tige de longueur ≤ 1, elle ne peut donc nous rapporter que son prix qui est forcément optimal, donc $r_0 = 0$ et $r_1 = p_1$;
— pour une tige de longueur 2, on peut soit la couper en 2, soit la laisser entière ; dans les deux cas, on peut se contenter d'aller consulter la table des prix, et la solution optimale est donc $r_2 = \max(p_2, p_1 + p_1)$;

5. Pour s'éviter des débats inutiles, considérez que "mémoïsation" et "mémorisation" sont synonymes.

— si la tige est de longueur 3, on a le choix entre la laisser entière, garder un morceau de longueur 1 intact, ou garder un morceau de longueur 2 intact ; la découpe optimale pour le reste a déjà été calculée, et il nous suffit donc d'aller consulter la table correspondante.
— ...

Cela signifie donc que pour chaque longueur de tige possible, il nous faut enregistrer la valeur de la découpe optimale qui nous servira dans la suite. On obtient donc l'algorithme 42.

Algorithme 42 : DÉCOUPEOPTIMALEPROGDYN(longueur, prix)

Entrées : une longueur naturelle de tige, un tableau de prix donnant pour chaque longueur possible le prix correspondant.
Sortie : le profit maximum réalisable.

```
1 profits ← prix;
2 pour chaque k allant de 2 à longueur faire
3 |   pour chaque i allant de 1 à k − 1 faire
4 |   |   profits[k] ← max(profits[k], prix[i] + profits[k − i]);
5 renvoyer profits[longueur];
```

Pour une longueur valant n, on consomme cette fois-ci un espace mémoire de l'ordre de $O(n)$, ce qui nous permet d'éviter les appels récursifs, les calculs inutiles, et de ramener la complexité à du $O(n^2)$.

7.4.5 Distance d'édition

Le problème suivant est basée sur une notion de distance entre chaînes de caractères ayant un grand nombre d'applications. On la retrouve entre autres dans la correction d'erreurs, et en bioinformatique où des variantes de cette distance sont utilisées pour comparer des génomes à l'aide d'"alignements".

Définition 37. La *distance d'édition*, ou *distance de Levenshtein*, entre deux chaînes de caractères est le plus petit nombre d'insertions, de suppressions et de modifications de caractères nécessaires à la transformation d'une de ces chaînes en l'autre.

Exemple 56. On peut transformer le mot "demain" en le mot "matin" en 11 étapes : il nous suffit de supprimer les 6 caractères de "demain" et d'écrire ensuite les 5 caractères du mot "matin". Mais cette valeur n'est pas la distance entre les deux mots, car on peut facilement trouver une solution plus courte en remarquant que les deux mots donnés se terminent par les deux mêmes caractères : une stratégie analogue nous permettrait donc d'obtenir une transformation en 7 étapes, en supprimant "dema" et en insérant ensuite "mat". Ceci n'est toujours pas optimal : il est possible de trouver une solution optimale en 3 étapes :

$$\text{demain} \to \text{emain} \to \text{main} \to \text{matin}.$$

On va suivre la stratégie de la sous-section 7.4.3 pour calculer cette distance.

Version naïve

Le problème du calcul de cette distance ne semble pas trivial ; une manière plus simple de s'y attaquer, qui marche en général pour l'élaboration d'algorithmes récursifs, est de se demander comment on pourrait résoudre ce problème en supposant que des sous-problèmes plus petits ont déjà été résolus. Par exemple, si l'on connaît une solution optimale pour deux chaînes de caractères X et Y, comment peut-on en déduire une solution optimale pour les paires $(X + c, Y)$, $(X + c, Y + d)$, et $(X, Y + d)$ (le $+$ représentant ici la concaténation de la chaîne X ou Y et des caractères c ou d) ?

Soit S et T deux chaînes de caractères, de longueurs respectives k et n. On peut calculer la distance d'édition entre ces deux chaînes en procédant de manière récursive. Les cas de base sont les suivants :

1. si S est vide, alors le seul moyen d'obtenir T est d'insérer tous les caractères de T, ce qui nous coûte donc $|T|$ opérations ;
2. si c'est T qui est vide, alors le seul moyen d'obtenir T est de supprimer tous les caractères de S, ce qui nous coûte donc $|S|$ opérations.

Dans le cas général où ni S ni T ne sont vides, on peut calculer la distance en essayant les trois opérations proposées et en conservant celle qui induira la distance totale la plus petite. On peut y arriver en ne modifiant que le dernier caractère de chacune des deux chaînes, et en appelant notre fonction récursivement sur le reste des deux chaînes pour calculer la distance totale. Comme le montre la Figure 7.2, on a trois choix possibles ;

1. si l'on supprime S_k, alors le coût total qu'induit ce choix est $1 + d(S^*, T)$;
2. si l'on supprime T_n, alors le coût total qu'induit ce choix est $1 + d(S, T^*)$;
3. si l'on met en correspondance S_k et T_n, alors le coût total qu'induit ce choix dépend de la comparaison de ces caractères : si $S_k = T_n$, alors le coût est $d(S^*, T^*)$, sinon le coût est $1 + d(S^*, T^*)$.

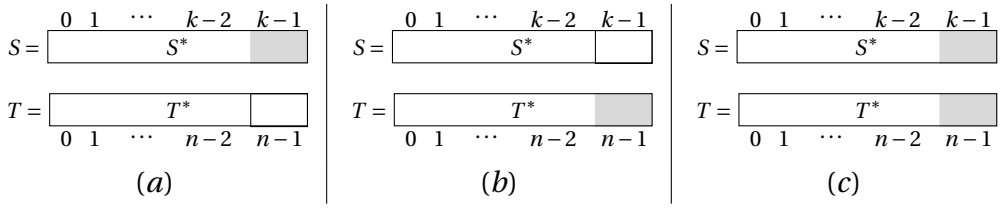


FIGURE 7.2 – Les trois choix dont on dispose pour comparer les derniers caractères de deux chaînes S et T : (a) supprimer $S[k-1]$, (b) supprimer $T[n-1]$, ou (c) supprimer les deux en mettant $S[k-1]$ en correspondance avec $T[n-1]$.

Ceci nous donne donc l'équation de programmation dynamique suivante pour calculer la distance de Levenshtein :

$$d(S,T) = \begin{cases} |S| & \text{si } |T| = 0, \\ |T| & \text{si } |S| = 0, \\ \min(1 + d(S^*, T), 1 + d(S, T^*), 1_{S_{k-1} \neq T_{n-1}} + d(S^*, T^*)) & \text{sinon.} \end{cases} \tag{7.2}$$

Et l'on déduit directement de cette équation l'algorithme 43.

Algorithme 43 : DISTANCEEDITIONNAÏVE(S, k, T, n)

Entrées : deux chaînes de caractères S et T et leurs longueurs (respectivement k et n)

Sortie : la distance d'édition entre S et T

```
1 si k = 0 alors renvoyer n;
2 si n = 0 alors renvoyer k;
3 option1 ← 1 + DISTANCEEDITIONNAÏVE(S, k − 1, T, n);
4 option2 ← 1 + DISTANCEEDITIONNAÏVE(S, k, T, n − 1);
5 option3 ← (S[k − 1] ≠ T[n − 1]) + DISTANCEEDITIONNAÏVE(S, k − 1, T, n − 1);
6 renvoyer min(option1, option2, option3);
```

Version avec table

L'algorithme naïf pour calculer la distance d'édition entre deux chaînes est particulièrement inefficace : chaque appel récursif peut en déclencher jusqu'à trois, ce qui nous donne au moins $3^{\min(k,n)}$ appels récursifs. On peut grandement améliorer les choses en stockant les résultats intermédiaires dans une matrice de coûts, que l'on remplira jusqu'à obtenir l'entrée qui nous intéresse.

Exemple 57. Soit S = "demain", et T = "matin". La matrice suivante montre la distance d'édition entre toutes les combinaisons possibles de préfixes de S et de suffixes de T :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

Pour construire cette matrice de coûts, on peut insérer un caractère vide factice au début de S et de T, pour représenter le cas où ces chaînes sont vides. Ensuite, pour chaque préfixe de longueur 0, 1, 2, … de S **et** de T, on stocke la distance d'édition entre ces préfixes que l'on obtient à l'aide de l'équation de programmation dynamique que l'on a établie. Ainsi :

— la première ligne de la matrice correspond à comparer la chaîne vide à tous les préfixes de T et correspondra donc à la séquence $(0, 1, 2, \ldots, |T|)$;
— la première colonne de la matrice correspond à comparer la chaîne vide à tous les préfixes de S et correspondra donc à la séquence $(0, 1, 2, \ldots, |S|)$;
— pour toutes les autres cases de la matrice, on calcule :

$$M_{ij} = \min(M[i-1][j]+1, M[i][j-1]+1, M[i-1][j-1]+1_{(}S[i] \neq T[j])).$$

L'algorithme 44 implémente cette approche, qui se déduit simplement de ce qu'on a expliqué jusqu'ici. Remarquons qu'on ne modifie pas S ni T, et qu'il y a un décalage entre les indices de la matrice et ceux des chaînes.

Algorithme 44 : DISTANCEEDITIONPROGDYN(S, T)

Entrées : deux chaînes de caractères S et T et leurs longueurs (respectivement k et n)
Sortie : la distance d'édition entre S et T

```
   /* initialiser la matrice de coûts avec les cas de base          */
1  k ← |S|;
2  n ← |T|;
3  coûts ← matrice de zéros de dimension (k+1) × (n+1);
4  pour chaque i allant de 0 à k faire coûts[i][0] ← i ;
5  pour chaque j allant de 0 à n faire coûts[0][j] ← j ;
   /* remplir la matrice de coûts à l'aide de la récurrence         */
6  pour chaque i allant de 1 à k faire
7      pour chaque j allant de 1 à n faire
8          option1 ← 1 + coûts[i−1][j];
9          option2 ← 1 + coûts[i][j−1];
10         option3 ← (S[i−1] ≠ T[j−1]) + coûts[i−1][j−1];
11         coûts[i][j] ← min(option1, option2, option3);
12 renvoyer coûts[k][n];
```

Exercice 12. La solution proposée pour calculer la distance d'édition possède une complexité en $O(nk)$ et une consommation mémoire en $O(nk)$. Comment pourrait-on obtenir une consommation en mémoire en $O(k+n)$?

7.4.6 Reconstruction des solutions optimales

Les algorithmes de programmation dynamique que l'on a présentés jusqu'ici ne nous fournissent pas une *solution* optimale, mais simplement leur *valeur*. Comment peut-on reconstruire les solutions correspondantes ?

Deux techniques viennent à l'esprit : reconstruire les choix optimaux réalisés à chaque étape, en parcourant la solution finale de la fin vers le début ; ou stocker explicitement, lors de la construction de la structure contenant les solutions optimales aux sous-problèmes, les décisions qui ont été prises. Illustrons ces techniques sur les exemples que nous avons déjà rencontrés.

Reconstruction d'une découpe optimale

Reprenons le problème de découpe d'une tige. Une manière de reconstruire la solution optimale consiste à enregistrer, en plus du coût de la solution optimale d'un sous-problème, la taille du dernier morceau de la découpe. Ainsi, pour reconstruire une solution optimale pour une tige de taille n, il nous suffirait d'afficher la longueur de la dernière pièce coupée dans une solution optimale, de retrancher cette longueur de la taille de la tige actuelle, et de réitérer l'opération jusqu'à ce qu'on l'indice de la case fournissant cette solution. On reconstruit ensuite une solution au départ de la fin pour une tige de taille n, en consultant le profit que l'on peut en tirer ainsi que la taille du morceau à couper pour obtenir le reste de la découpe et en réitérant ce processus jusqu'à ce que toutes les pièces aient été obtenues.

Exemple 58. Supposons que la longueur maximale d'une tige est 10, et que les prix sont fournis par le tableau suivant :

longueur	0	1	2	3	4	5	6	7	8	9	10
prix	0	1	5	8	9	10	17	17	20	24	30

Après quelques modifications (expliquées plus bas), l'algorithme 42 fournit la solution optimale pour chaque longueur de tige jusqu'à 10, et également la taille du dernier morceau de cette découpe :

longueur	0	1	2	3	4	5	6	7	8	9	10
prix	0	1	5	8	9	10	17	17	20	24	30
profit	0	1	5	8	10	13	17	18	22	25	30
taille_dernière_pièce	0	1	2	3	2	2	6	1	2	3	10

Ces deux informations nous permettent de déduire une découpe optimale explicite dans tous les cas ; par exemple, si la longueur vaut :

— 9, alors le dernier morceau utilisé est de taille 3, et une découpe optimale est donnée par 9 = 3 + 6 (sans autre découpe puisque 6 est une longueur optimale) ;
— 5, alors le dernier morceau utilisé est de taille 2, et une découpe optimale est donnée par 5 = 2 + 3 (sans autre découpe puisque 3 est une longueur optimale).

Pour obtenir ces informations, il faut légèrement modifier l'algorithme 42 afin qu'il stocke, lors de l'étape de découpe, la taille de tige qui permet d'obtenir le résultat optimal. La modification est incorporée à l'algorithme 45.

Algorithme 45 : DÉCOUPEOPTIMALEPROGDYNSOL(longueur, prix)

Entrées : une longueur naturelle de tige, un tableau de prix donnant pour chaque longueur possible le prix correspondant.
Sortie : les profits maximaux réalisables pour chaque longueur de tige, ainsi que la longueur du dernier morceau pour chaque découpe optimale.

```
1  profits ← prix;
2  taille_dernière_pièce ← tableau d'entiers de longueur |prix|;
3  pour chaque k allant de 2 à longueur faire
4      bénéfice ← prix[k];
5      taille_dernière_pièce[k] ← k;
6      pour chaque i allant de 1 à k − 1 faire
7          si prix[i] + profits[k − i] > bénéfice alors
8              bénéfice ← prix[i] + profits[k − i];
9              taille_dernière_pièce[k] ← i;
10     profits[k] ← bénéfice;
11 renvoyer (profits, taille_dernière_pièce);
```

Un algorithme trivial permet ensuite de reconstruire la solution à l'aide de ces deux tableaux : tant que la longueur est positive, on lui retranche la taille de la dernière pièce dans une découpe optimale et on réitère l'opération.

Reconstruction d'une séquence optimale de transformations pour la distance d'édition

Reprenons le problème du calcul de la distance d'édition entre deux chaînes. Pour reconstruire une solution optimale, on pourrait en plus des coûts stocker le choix qui nous a mené à chacune des cases. Cette solution est inefficace, car elle consomme $O(kn)$ cases et seul un chemin optimal nous intéresse.

Au lieu de cela, si l'on dispose de la matrice de coûts, on peut reconstruire les choix qui nous ont mené à la solution optimale dont la valeur est donnée en coûts$[k][n]$: l'étape précédente qui nous a mené en cette case consiste en la transition de moindre coût au départ de coûts$[k-1][n-1]$, de coûts$[k-1][n]$, ou de coûts$[k][n-1]$. Il nous suffit donc de comparer les valeurs de coûts$[k][n-1]+1$, coûts$[k-1][n]+1$, et coûts$[k-1][n-1]+x$, où x est le coût d'une différence entre $S[k-1]$ et $T[n-1]$.

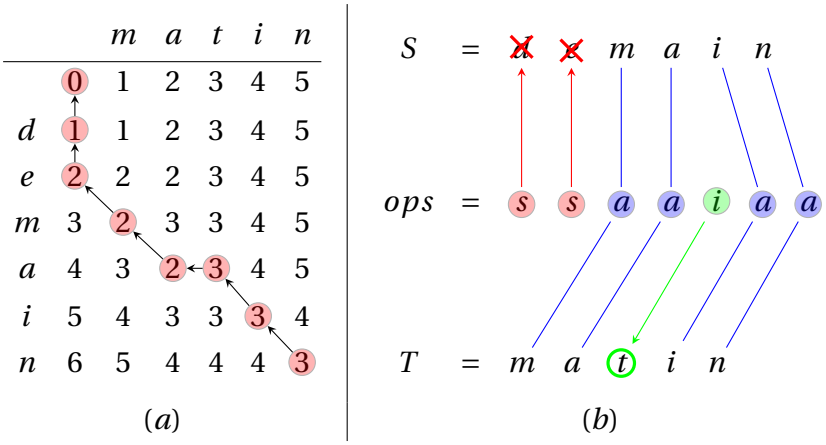
De la même manière, le choix que l'on fait pour remonter d'une case vers une case précédente nous indique quelle opération a été appliquée dans la matrice de coûts, où l'on réduit S et T à deux chaînes vides :

- $\nwarrow$: supprime le dernier caractère de S et le dernier caractère de T ;
- $\leftarrow$: supprime le dernier caractère de T ;
- $\uparrow$: supprime le dernier caractère de S.

La reconstruction d'une séquence optimale de transformations se fera au départ de S (ou de T), en suivant le chemin à l'envers et en associant à chaque symbole une opération :

- $\nwarrow$: **a**ssocie le caractère actuel de S et le caractère actuel de T ;
- $\leftarrow$: **s**upprime le caractère actuel de S ;
- $\uparrow$: **i**nsère un caractère à l'endroit actuel dans S (équivaut à une suppression dans T).

Exemple 59. Reprenons la matrice de coûts de l'exemple 57. La distance entre "demain" et "matin" est de 3 comme l'indique la dernière case de la matrice ; pour reconstruire un chemin optimal, on reconstruit les décisions optimales qui nous ont mené à cette case, ce qui donne le chemin montré en (a) :



Le chemin suivi à l'envers donne la séquence $\uparrow\uparrow\nwarrow\nwarrow\leftarrow\nwarrow\nwarrow$, ou "ssaaiaa" (pour **s**uppression (dans S), **a**ssociation, et **i**nsertion (dans S)). Ainsi, l'application de cette séquence montrée en (b) supprime les deux premiers caractères de S, associe les deux suivants aux deux premiers de T, insère la lettre "t" dans S, et associe ensuite les deux derniers caractères de S et de T.

L'algorithme 46 nous permet au départ d'une case de reconstruire la meilleure transition, et donc de reconstruire un chemin optimal sur base de la matrice de coûts.

Algorithme 46 : MEILLEURPARENT(matrice, i, j, coût_différence)

Entrées : une matrice de coûts, un numéro i de ligne, un numéro j de colonne, et le coût d'une différence entre deux caractères.

Sortie : les coordonnées du parent de matrice$[i][j]$ de coût minimum.

```
// enregistrer les transitions possibles et leur coût
transitions ← tableau_associatif();
si i ≥ 1 et j ≥ 1 alors transitions[(i − 1, j − 1)] ← matrice[i − 1][j − 1] +
 coût_différence ;
si i ≥ 1 et j ≥ 0 alors transitions[(i − 1, j)] ← matrice[i − 1][j] + 1;
si i ≥ 0 et j ≥ 1 alors transitions[(i, j − 1)] ← matrice[i][j − 1] + 1 ;
// sélectionner le minimum et renvoyer ses coordonnées
coordonnées ← NIL;
minimum_actuel ← +∞;
pour chaque coords, valeur ∈ transitions faire
    si valeur < minimum_actuel alors
        minimum_actuel ← valeur;
        coordonnées ← coords;
renvoyer coordonnées;
```

L'algorithme de Bellman-Ford

L'algorithme de Bellman-Ford, qui calcule les plus courts chemins à origine unique dans un graphe admettant des arcs négatifs, est un algorithme de programmation dynamique. Pour le voir, appliquons le schéma présenté plus haut pour le reconstruire.

Il nous faut identifier une structure de sous-solution optimale que l'on peut calculer de manière récursive. Pour ce faire, remarquons qu'un plus court chemin entre deux sommets arbitraires du graphe contient au plus $|V|$ sommets. Si l'on veut atteindre le sommet t à partir du sommet s, on peut obtenir le plus court chemin Ceci n'est pas compliqué à réaliser : on peut calculer la longueur d'un plus court chemin entre s et t en tentant de passer par tous les autres sommets du graphe et en conservant l'intermédiaire qui nous fournit la plus petite distance. Plus formellement, on a :

$$d(s,t) = \min_{v \in V(G)} (d(s,v) + d(v,t)).$$

Bien entendu, ceci implique de connaître à la fois $d(s,v)$ et $d(s,t)$, ce qui implique d'effectuer des appels récursifs pour ces deux valeurs. Quelle sera la condition d'arrêt de ces appels récursifs ? C'est simple : quand $s = t$, on renvoie 0, et quand $(s,t) \in A(G)$ (ou $\{s,t\} \in E(G)$), on renvoie son poids.

L'algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est un algorithme de programmation dynamique. Pour le voir, appliquons le schéma présenté plus haut pour le reconstruire.

Sous-chemins optimaux On appelle *sommet intermédiaire* d'un chemin de i à j un sommet différent de i et de j apparaissant dans ce chemin. Supposons pour simplifier que les sommets du graphe sont numérotés de 1 à n, et considérons le sous-ensemble formé des k premiers sommets du graphe pour n'importe quelle valeur de k. Soit P un plus court chemin entre deux sommets i et j du graphe dont les sommets intermédiaires sont tous dans l'ensemble $\{1,2,\ldots,k\}$; on distingue deux cas :

1. soit k n'est **pas** un sommet intermédiaire de ce chemin, auquel cas P est également un plus court chemin entre i et j dont tous les sommets intermédiaires sont dans $\{1,2,\ldots,k\}$;
2. soit k est un sommet intermédiaire de ce chemin, auquel cas on peut décomposer P en deux chemins P_1 de i à k et P_2 de k à j. P_1 et P_2 sont également des plus courts chemins, et comme P est simple, k ne peut pas apparaître plus d'une fois ; donc P_1 et P_2 ne contiennent que des sommets dans $\{1,2,\ldots,k-1\}$.

7.5 Branch-and-bound

Pour résoudre certains problèmes, il arrive que l'on n'ait pas d'autre idée (ou d'autre choix) que de passer en revue toutes les possibilités : on qualifie cette approche de *recherche exhaustive*. Pour les problèmes d'optimisation, il existe une variante plus intelligente qui consiste à se débarrasser des solutions partielles qui ne peuvent pas mener à une solution meilleure que celle qu'on connaît actuellement. Cette approche, inventée par Land et Doig [7], est connue sous le nom de "*branch-and-bound*". Nous l'illustrerons sur des problèmes de minimisation, mais le raisonnement est analogue dans le cas des problèmes de maximisation.

Afin de comprendre son fonctionnement, partons de la construction des solutions dans la recherche exhaustive, et plaçons-nous dans une situation où le problème que l'on veut résoudre exige de la recherche exhaustive qu'elle passe en revue toutes les permutations de n éléments et sélectionne la "meilleure" d'entre elles. On générerait alors toutes les permutations commençant par 1, puis toutes celles commençant par 2, … et ainsi de suite jusqu'à n. Lorsque l'on a fixé la valeur de la première position, on essaie tous les choix possibles pour la seconde, puis pour la troisième, et ainsi de suite jusqu'à ce que la séquence soit complète. La Figure 7.3 illustre le procédé.

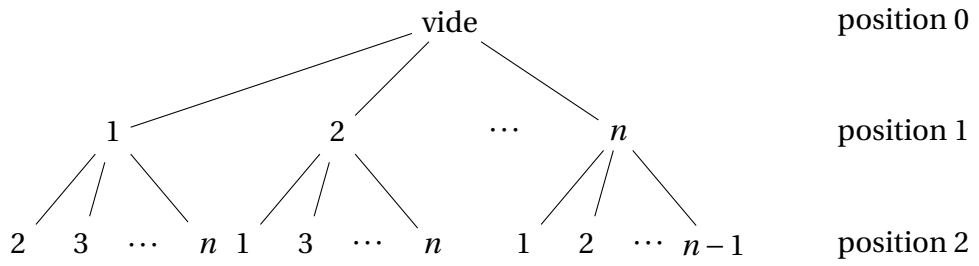


FIGURE 7.3 – L'arbre de construction de toutes les permutations de $\{1,2,\ldots,n\}$.

Un algorithme de *branch-and-bound* suivra la même exploration, mais en sélectionnant de manière intelligente les branches de l'arbre qui valent la peine d'être explorées : plus on

détecte tôt qu'une solution n'est pas prometteuse, et plus on peut éliminer de candidats inutiles. Pour parvenir à ces fins, l'algorithme utilisera :

1. une fonction de majoration, qui nous donne une borne supérieure sur la qualité d'une solution optimale ;
2. une fonction de minoration, qui nous donne une borne inférieure sur la qualité d'une solution optimale.

Ces deux fonctions nous sont déjà utiles pour limiter l'espace de recherche d'une solution ; mais le *branch-and-bound* pousse les choses plus loin, en utilisant ces fonctions à chaque examen d'une nouvelle branche de l'arbre de recherche, en exploitant la minoration comme suit : si la solution partielle actuelle S nous mène à explorer une branche pour laquelle la minoration donne une valeur supérieure au coût de S, alors on sait qu'explorer cette branche est inutile puisqu'elle ne nous permettra pas d'obtenir une meilleure solution que S.

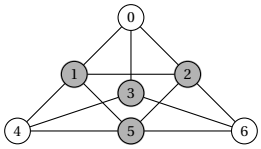
Examinons la façon dont on peut tirer parti du *branch-and-bound* sur quelques exemples.

7.5.1 Application : VERTEX COVER

Le problème NP-complet VERTEX COVER est un grand classique parmi les problèmes d'optimisation. Il demande de trouver un ensemble de sommets de taille minimale dans un graphe qui "couvre" toutes les arêtes du graphe ; autrement dit, chaque arête doit posséder au moins une extrémité dans cet ensemble.

VERTEX COVER
Entrées : un graphe (non orienté, non pondéré) $G = (V, E)$, une taille $K \in \mathbb{N}$.
Question : existe-t-il un sous-ensemble $U \subseteq V$ avec $|U| \leq K$ et tel que $\forall\ \{u, v\} \in E : |\{u, v\} \cap U| \geq 1$?

Exemple 60. Voici un graphe et une solution valide de taille 4 pour VERTEX COVER :



Comme de nombreux autres problèmes difficiles, VERTEX COVER ne semble pas particulièrement difficile à résoudre. Cependant, il reste NP-complet même si l'on suppose G cubique, ou planaire de degré maximum 3 ; la meilleure approximation a un taux de 2, et il est très peu probable qu'il soit possible de faire mieux (on sait en tout cas qu'on ne peut pas faire mieux que $\sqrt{2} \approx 1.4142\ldots$).

Il est facile d'écrire un algorithme de recherche exhaustive pour résoudre VERTEX COVER : on passe en revue tous les sous-ensembles de 1, 2, …, $|V|$ sommets jusqu'à ce que l'on en trouve un qui couvre toutes les arêtes. Bien entendu, la complexité en $O(2^{|V|})$ de cet algorithme sera prohibitive ; et l'on verra que même en recherche exhaustive, il est possible d'obtenir de meilleurs algorithmes.

Bornes sur la taille d'une solution optimale

Les couplages maximaux vus en sous-section 6.4.2 permettent d'obtenir une minoration et une majoration sur la taille d'une solution optimale à VERTEX COVER. Commençons par la

majoration.

Lemme 7.5.1. Soit G une instance de VERTEX COVER, et M un couplage maximal pour G; alors les sommets couverts par M constituent une solution valide. Donc $|\mathsf{Opt}(G)| \leq 2|M|$.

Démonstration. Si M est maximal, on ne peut par définition pas lui ajouter d'arête. Dès lors, chaque arête de G possède une extrémité dans M, et sélectionner tous les sommets couverts par M permet donc de couvrir toutes les arêtes de G. □

Lemme 7.5.2. Soit G une instance de VERTEX COVER, et M un couplage maximal pour G; alors toute solution valide pour G doit contenir une extrémité de chaque arête de M. Donc $|\mathsf{Opt}(G)| \geq |M|$.

Démonstration. Si M est maximal, ses sommets couvrent toutes les arêtes du graphe; on doit donc sélectionner au moins une extrémité de chaque arête de M, puisqu'une arête de M n'ayant aucune extrémité sélectionnée n'est pas couverte. □

Remarquons que ces deux bornes sont déjà exploitables sans même parler de *branch-and-bound* : premièrement, elles permettent à la recherche exhaustive de resserrer l'intervalle de recherche, puisqu'une fois un couplage maximal trouvé M, il "suffit" de n'examiner "que" les sous-ensembles de taille $|M|, |M|+1, \ldots, 2|M|$ plutôt que tous les sous-ensembles. Deuxièmement, elles nous donnent directement une 2-approximation pour VERTEX COVER, puisque la taille d'une solution optimale est toujours comprise entre $|M|$ et $2|M|$; il ne nous reste plus qu'à savoir comment trouver un couplage maximal. Il est possible de trouver un couplage maximum en $O(\sqrt{|V|}|E|)$ à l'aide d'un algorithme relativement complexe [8], mais trouver un couplage maximal est plus simple : on part d'un couplage vide, auquel on rajoute à chaque étape, tant que c'est possible, une arête arbitraire du graphe, dont on supprime ensuite les extrémités. Comme l'algorithme se termine quand il n'y a plus d'arêtes, le couplage obtenu est forcément maximal. L'algorithme 47 implémente cette approche et s'exécute en $O(|E|)$.

Algorithme 47 : COUPLAGEMAXIMAL(G)

Entrées : un graphe $G = (V, E)$ non orienté, non pondéré.
Sortie : un couplage maximal pour G.

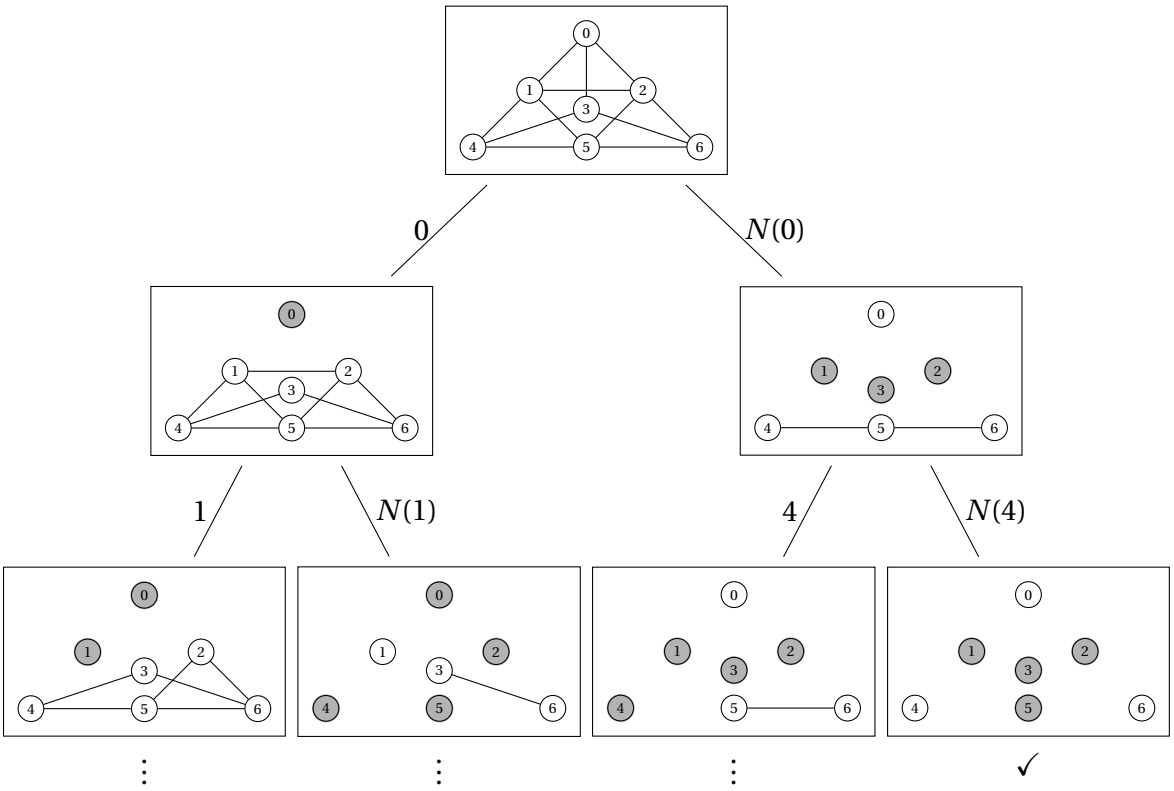
```
1 M ← ∅;
2 G' ← G;
3 tant que G.nombre_arêtes() ≠ 0 faire
4 |   M ← M ∪ une arête arbitraire {u, v} de G;
5 |   G.retirer_sommets({u, v});
6 G ← G';
7 renvoyer M;
```

Revenons à présent à VERTEX COVER, et commençons par écrire l'algorithme de recherche exhaustive. Nous pourrions générer directement tous les sous-ensembles qui nous intéressent, mais nous allons les construire pas à pas, en décidant à chaque étape d'ajouter ou non un sommet v à notre solution en cours de construction, afin de pouvoir ensuite interrompre la construction de solutions partielles en cours de route. Il y a deux options :

1. soit on sélectionne v, ce qui couvre ses arêtes incidentes qu'on retire donc du graphe ;
2. soit on ne le sélectionne pas ; mais alors, il nous faut sélectionner tous ses voisins pour que les arêtes incidentes à v soient toutes couvertes ; on supprime ensuite les arêtes incidentes à $N(v)$ du graphe.

Dans les deux cas, on relance l'algorithme récursivement sur ce qui reste du graphe, avec comme condition triviale d'arrêt l'absence d'arêtes.

Exemple 61. Voici les premières étapes du déroulement de la recherche exhaustive sur le graphe de l'exemple 60 :



L'algorithme 48 implémente cette approche.

Le *branch-and-bound* va exploiter les bornes que nous avons prouvées de la façon suivante :
— on sélectionne un U initial correspondant aux extrémités d'un couplage maximal, afin d'avoir une solution initiale de taille $\leq |V|$;
— si l'on trouve à un moment une solution dont la taille égale la minoration, alors elle est optimale et on peut s'arrêter ;
— avant de lancer la recherche sur le graphe G' (obtenu en sélectionnant soit v, soit $N(v)$), on calcule la valeur de la minoration sur G' : si cette valeur mène à une solution moins bonne que notre U actuel, on ne lance pas l'exploration récursive.

Ces améliorations transforment l'algorithme 48 en l'algorithme 49.

Remarquons que la complexité de l'algorithme 49 ne s'en retrouve pas améliorée. En fait, elle est même un peu plus élevée, puisqu'en plus de l'exploration au pire cas de tous les sous-ensembles, on calcule deux couplages maximaux par sommet potentiel. Cependant, on espère que ces bornes nous permettront d'éliminer suffisamment de candidats pour que l'approche *branch-and-bound* soit plus performante en pratique.

Algorithme 48 : VCEXHAUSTIF(G)

Entrées : un graphe $G = (V, E)$ non orienté, non pondéré.
Sortie : un ensemble $U \subseteq V$ de taille minimale couvrant toutes les arêtes de G.

```
1  si G.nombre_arêtes() = 0 alors renvoyer ∅;
2  U ← V;
3  pour chaque v dans G.sommets() de degré > 0 faire
       // sélectionner v ...
4      arêtes_ôtées ← G.arêtes_incidentes(v);
5      G.supprimer_arêtes(arêtes_ôtées);
6      A ← {v} ∪ VCEXHAUSTIF(G);
7      si |A| < |U| alors U ← A;
8      G.ajouter_arêtes(arêtes_ôtées);
       // ... ou sélectionner N(v)
9      voisins ← G.voisins(v);
10     arêtes_ôtées ← G.arêtes_incidentes(voisins);
11     G.supprimer_arêtes(arêtes_ôtées);
12     B ← voisins ∪ VCEXHAUSTIF(G);
13     si |B| < |U| alors U ← B;
14     G.ajouter_arêtes(arêtes_ôtées);
15 renvoyer U;
```

Il y a un certain nombre d'améliorations que l'on peut apporter à l'algorithme 49 :

- au lieu de générer une solution en ne rajoutant qu'un élément à chaque fois, on pourrait directement parcourir tous les sous-ensembles dont la taille est celle de la minoration ;
- certains sous-cas sont solubles en temps polynomial, comme celui des graphes bipartis ou des graphes complets : on pourrait donc immédiatement renvoyer la solution optimale correspondante ;
- certaines sous-instances pourraient être générées plusieurs fois, puisque certaines solutions partielles ne diffèrent que par l'ordre dans lequel on les construit ; si on ne peut pas l'éviter, on pourrait avoir recours à la mémoïsation comme dans la programmation dynamique ;
- on pourrait examiner les sommets par degré décroissant, puisqu'ils sont plus rentables ;
- ...

7.6 Pour aller plus loin

Distance d'édition. Comme pour beaucoup de problèmes que l'on a vus dans ce cours, il est naturel de se demander s'il est possible de calculer la distance d'édition entre deux chaînes de longueur n avec une meilleure complexité que $O(n^2)$. La réponse est non, comme l'ont récemment prouvé Backurs et Indyk [11]. Pour être plus exact, il faut un peu nuancer cette affirmation : comme la plupart des résultats de complexité, leur résultat est une preuve par contradiction de la forme : "**si** il existait un algorithme de complexité ..., **alors** une certaine hypothèse serait réfutée", sachant qu'il est très probable (mais non prouvé) que l'hypothèse soit vraie. C'est pourquoi de nombreux articles présentent prudemment des résultats

Algorithme 49 : VCBranchAndBound(G)

Entrées : un graphe $G = (V, E)$ non orienté, non pondéré.
Sortie : un ensemble $U \subseteq V$ de taille minimale couvrant toutes les arêtes de G.

```
1  si G.nombre_arêtes() = 0 alors renvoyer ∅;
2  M ← CouplageMaximal(G);
3  U ← les extrémités des arêtes de M;
4  pour chaque v dans G.sommets() de degré > 0 faire
       // sélectionner v ...
5      arêtes_ôtées ← G.arêtes_incidentes(v);
6      G.supprimer_arêtes(arêtes_ôtées);
7      M' ← CouplageMaximal(G);
8      si |M'| + 1 < |U| alors                          // solution prometteuse
9          A ← {v} ∪ VCBranchAndBound(G');
10         si |A| < |U| alors U ← A;
11     G.ajouter_arêtes(arêtes_ôtées);
12     si |U| = |M| alors renvoyer U;                    // solution optimale
       // ... ou sélectionner N(v)
13     voisins ← G.voisins(v);
14     arêtes_ôtées ← G.arêtes_incidentes(voisins);
15     G.supprimer_arêtes(arêtes_ôtées);
16     M' ← CouplageMaximal(G);
17     si |M'| + |voisins| < |U| alors                   // solution prometteuse
18         B ← voisins ∪ VCBranchAndBound(G);
19         si |B| < |U| alors U ← B;
20     G.ajouter_arêtes(arêtes_ôtées);
21     si |U| = |M| alors renvoyer U;                    // solution optimale
22 renvoyer U;
```

sous la forme “Si P≠ NP, alors . . . ”.

Arbres de recherche. La technique utilisée dans l'algorithme 48 pour résoudre VERTEX COVER est qualifiée d'“arbre de recherche”, et forme la base de techniques beaucoup plus avancées et performantes. On la retrouve notamment dans la théorie des *algorithmes paramétrés*, dont vous entendrez peut-être parler dans des cours plus avancés.

Bibliographie

[1] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, ET C. STEIN, *Introduction to Algorithms*, MIT Press, 3ème ed., 2009.

[2] M. R. GAREY ET D. S. JOHNSON, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[3] G. DÓSA ET J. SGALL, *First fit bin packing: A tight analysis*, dans 30th International Symposium on Theoretical Aspects of Computer Science (STACS), édité par N. Portier et T. Wilke, vol. 20 de LIPIcs, Kiel, Allemagne, Février 2013, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pages 538–549.

[4] V. V. VAZIRANI, *Approximation algorithms*, Springer, 2001.

[5] A. BLUM, T. JIANG, M. LI, J. TROMP, M. YANNAKAKIS, ET M. YANNAKAKIS, *Linear approximation of shortest superstrings*, Journal of the ACM, 41 (1994), pages 630–647.

[6] S. DREYFUS, *Richard Bellman on the birth of dynamic programming*, Operations Research, 50 (2002), pages 48–51.

[7] A. H. LAND ET A. G. DOIG, *An automatic method of solving discrete programming problems*, Econometrica, 28 (1960), pages 497–520.

[8] S. MICALI ET V. V. VAZIRANI, *An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs*, dans 21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980, IEEE Computer Society, 1980, pages 17–27.

[9] N. CHRISTOFIDES, *Worst-case analysis of a new heuristic for the travelling salesman problem*, Tech. Rep. 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Février 1976.

[10] M. KARPINSKI, M. LAMPIS, ET R. SCHMIED, *New inapproximability bounds for tsp*, Journal of Computer and System Sciences, 81 (2015), pages 1665–1677.

[11] A. BACKURS ET P. INDYK, *Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)*, SIAM Journal on Computing, 47 (2018), pages 1087–1097.