Méthodes Algorithmiques

21 mai 2024

1 Algorithmes Gloutons

Un **problème d'optimisation** est un problème dont la sortie doit correspondre à une valeur optimale selon un critère donné en fonction de la valeur en entrée.

Le caractère optimal de la sortie en fonction de l'entrée est déterminé par la minimisation ou maximisation d'une fonction. Un problème d'optimisation est caractérisé par la donnée de cette fonction (*fonction de coût* dans le cas d'une minimisation) et des contraintes à satisfaire.

Exemple. Les problèmes ci-dessous sont des exemples de problèmes d'optimisation :

- Rendu de Monnaie :
  - **Entrée :** Une valeur v et un système S de pièces de monnaie (exemple : pour les euros, $S = (1, 2, 5, 10, 20, 50, 100, 200, 500)$)
  - **Sortie :** Un ensemble de pièces dans le système S dont le total vaut v, avec un nombre de pièces minimal
- Problème du Sac à Dos :
  - **Entrée :** Un poids limite p et un ensemble E d'objets e_i, chacun ayant un poids p_i et une valeur v_i
  - **Sortie :** Une partie de E de poids total inférieur à p maximisant la valeur totale
- Problème du Voyageur de Commerce :
  - **Entrée :** Un ensemble de villes toutes reliées entre elles, et la donnée des distances entre chaque paire de villes
  - **Sortie :** Un circuit passant par toutes les villes, minimisant la distance totale parcourue
- Plus Court Chemin Dans le Plan :
  - **Entrée :** Ensemble E de points du plan, muni de la distance euclidienne $d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$, point de départ a et point d'arrivée b
  - **Sortie :** Plus court chemin entre a et b passant par des points de e

Dans les cas ci-dessous, pour une entrée donnée, l'ensemble des solutions possibles respectant les contraintes, parmi lesquelles on cherche la solution optimale, est fini. On peut techniquement envisager de trouver la solution qui optimise la fonction de coût par recherche exhaustive. Néanmoins, une recherche exhaustive peut s'avérer coûteuse (explosion combinatoire).

Certaines méthodes existent pour calculer l'optimum global en un temps raisonnable (programmation dynamique) ; néanmoins, trouver une façon d'utiliser ces méthodes pour un problème donné n'est pas toujours trivial, voire impossible.

Définition

Pour un problème donné, un **algorithme glouton** est un algorithme qui utilise une heuristique consistant à calculer une solution à l'aide de plusieurs calculs, étape par étape, d'optimisations locales.

Exemple. On considère le problème de rendu de monnaie. Un algorithme glouton pour résoudre le problème peut consister à rendre une pièce de valeur la plus élevée inférieure à la valeur à rendre, et réitérer l'opération jusqu'à ce que la somme à rendre soit nulle.

Attention

Dans de nombreux cas, le résultat renvoyé par un algorithme glouton ne correspond pas au résultat optimal recherché dans la spécification du problème.

Exercice 1

On considère le système de pièces `[1;3;4]`. Trouver une valeur sur laquelle l'application de l'algorithme glouton donné précédemment ne renvoie pas le résultat optimal.

On peut étudier les conditions sous lesquelles des algorithmes gloutons peuvent permettre d'obtenir des résultats optimaux. En particulier, pour le problème de rendu de monnaie, avec le système de pièces habituel (1, 2, 5, 10, 20, 50, 100, 200, 500), l'algorithme glouton fournit toujours un résultat optimal.

Bien que les algorithmes gloutons ne garantissent pas que l'on puisse obtenir un résultat exactement optimal, le fait qu'ils permettent de calculer un résultat proche du résultat optimal en peu de temps fait qu'ils sont souvent utilisés en pratique.

Exercice 2

Déterminer des stratégies gloutonnes pour le problème du sac à dos et le problème du plus court chemin dans le plan. Trouver des entrées pour lesquelles le résultat obtenu avec cette stratégie n'est pas optimal.

Exercice 3

On considère un ensemble de séances de cinéma dans une journée, dont on connaît les heures de début et de fin. On cherche à trouver un sous-ensemble de cet ensemble de séances qui maximise le nombre de films vus dans la journée, sans incompatibilité entre les horaires. La stratégie gloutonne suivante est envisageable :

— Classer les films selon une valuation donnée

— Dans la liste triée des films : prendre le premier film, et éliminer de la liste les films incompatibles avec le film sélectionné

— Réitérer l'opération jusqu'à avoir vider la liste des séances de films.

1) Montrer que la stratégie gloutonne n'est pas optimale lorsque l'on prend la durée ou l'heure de début du film comme valuation.

2) Trouver une valuation pour laquelle la stratégie gloutonne est optimale.

Diviser Pour Régner

On s'intéresse dans les deux sections suivantes à des méthodes pour résoudre des problèmes à travers la résolution de sous-problèmes, correspondant à des instances du problème initial avec des entrées de taille inférieure.

La stratégie **Diviser pour Régner** est une approche consistant à résoudre un problème, sur une entrée ne correspondant pas à un *cas de base*, en résolvant **sous-problèmes indépendants**.

Parmi les algorithme utilisant la stratégie Diviser pour Régner, on peut citer :

- L'exponentiation rapide
- La recherche dichotomique
- Le tri fusion
- Recherche de Deux points les plus proches dans le plan

1.1 Schéma Algorithmique et Complexité

Un algorithme de type Diviser pour Régner est divisé en plusieurs étapes :

- **Diviser :** on définit les sous-problèmes dont la résolution permet de résoudre le problème initial
- **Régner :** On résout les sous-problèmes
- **Rassembler :** On combine les solutions de chaque sous-problème pour trouver la solution du problème initial.

La résolution des sous-problèmes peut se faire de plusieurs façon : soit on se situe dans un cas de base, soit on effectue la résolution du sous-problème en effectuant encore au moins une division en sous-problèmes. Ce schéma peut en particulier s'exprimer à l'aide d'une définition récursive de la fonction permettant de résoudre le problème.

L'expression du schéma algorithmique permet d'avoir une expression récursive de la complexité sur une entrée de taille n, si les sous-problèmes sont tous de taille $\frac{n}{k}$. L'utilisation de ce schéma récursif permet de trouver des bornes à la complexité algorithmique d'un algorithme de type Diviser pour Régner.

1.2 Tri Fusion

Rappel : en OCaml, on suppose que l'on a défini :

- Une fonction `division` qui prend en entrée une liste et renvoie un couple de listes de tailles $\lfloor\frac{n}{2}\rfloor$ et $\lceil\frac{n}{2}\rceil$
- Une fonction `fusion` qui prend en entrée un couple de listes triées en renvoie la fusion triée des deux listes

La fonction `tri_fusion` qui prend en entrée une liste et renvoie la même liste triée est définie de la façon suivante à partir des fonctions `division` et `fusion` :

- Si la liste `l` en entrée est de taille 0 ou 1, renvoyer la liste telle quelle

— Sinon, renvoie `fusion (tri_fusion l1) (tri_fusion l2)` avec `(l1,l2) = division l`

La division correspond à la partie Diviser ; le tri fusion appelé récursivement à la partie Régner ; la fusion à la partie Rassembler.

1.3 Deux Points les Plus Proches dans le Plan

On considère le problème suivant :

— **Entrée :** un ensemble E de n points dans le plan

— **Sortie :** la plus petite distance entre deux points de E

1) Proposer une stratégie naïve pour résoudre le problème. Quelle est sa complexité ?

2) Proposer une stratégie de type Diviser pour Régner pour résoudre le problème.

2 Programmation Dynamique

Lorsque l'on utilise une stratégie de type Diviser pour Régner, les sous-problèmes sont indépendants : la résolution de deux sous-problèmes P_1 et P_2 n'induit pas, de chaque côté, de résolution d'un sous-problème commun. On peut donc, par exemple, définir un algorithme de façon récursive sans avoir de calculs superflus.

Ce n'est pas toujours le cas : on peut par exemple avoir des algorithmes sur des entrées de taille n, dont la solution peut s'exprimer en fonction des solutions de tous les sous-problèmes de taille $n-1$. Celles-ci peuvent elles-mêmes s'exprimer en fonction des solution des sous-problèmes de taille $n-2$, qui sont sollicitées dans les calculs pour les instances de taille $n-1$, ce qui entraînera des appels récursifs superflus avec un algorithme récursif naïf.

Ces calculs peuvent entraîner des changements d'ordre de grandeur du coût algorithmique par rapport à une version optimale. Par exemple, dans l'implémentation récursive naïve de la suite de Fibonacci ($F(n) = n$ si $n \leq 1$ et $F(n) = F(n-1) + F(n-2)$ sinon), on appelle plusieurs fois le calcul de $F(k)$ dans les différents appels récursifs, ce qui induit un coût exponentiel. Or on sait qu'une version itérative simple de la fonction a un coût linéaire, en calculant une seule fois chaque valeur nécessaire.

2.1 Schéma Algorithmique

La **Programmation Dynamique** est une approche consistant à décomposer les problèmes en sous-problèmes non nécessairement indépendant, en évitant les calculs redondants, en stockant en mémoire les solutions des sous-problèmes de taille inférieure.

Exemple. La version suivante de la fonction `fibo` utilise un tableau pour stocker à l'indice i la valeur $F(i)$:

```
int fibo(int n) {
    int* F = malloc(n*sizeof(int));
    F[0] = 0;
    F[1] = 1;
    for(int i=2; i < n; i++)
        F[i] = F[i - 1] + F[i - 2];
    return F[n - 1];
}
```

Remarque. On a vu en début d'année une version de la suite de Fibonacci dans laquelle on utilisait deux variables seulement pour stocker les valeurs utilisées à chaque itération de boucle : on pouvait procéder ainsi car on savait quelles étaient les valeurs à utiliser pour le calcul de $F(n)$. On ne le sait pas forcément en règle générale, on aura donc une structure stockant toutes les solutions pour les instances de taille inférieure à celle de l'entrée initiale.

Remarque. On utilise ici un tableau pour stocker les solutions, car les entrées sont des entiers et leurs solutions peuvent être stockées aux indices correspondant. Dans le cas où les instances du problème ne sont pas des entiers, on pourra utiliser une autre structure adéquate (dictionnaire par exemple).

La programmation dynamique est notamment utilisée pour des problèmes d'optimisation. On recherche une **sous-structure optimale :** un optimum pour une instance d'un problème peut s'exprimer en fonction des optima pour des instances plus petites.

Pour les problèmes d'optimisation, on peut citer comme stratégies :

— L'**approche exhaustive** dans laquelle on va explorer toutes les possibilités de sortie en fonction de l'entrée. On aboutit généralement avec des coûts élevés (exponentiel en fonction de la taille de l'entrée) ;

— L'**approche récursive** naïve qui va exprimer la solution d'un problème en fonction des solutions des sous-problèmes, sans prendre en compte les potentiels recalculs de solution, ce qui peut également aboutir à des coûts exponentiels

— Les **stratégies gloutonnes** qui ont un coût généralement faible mais donnent une solution approchée du problème.

L'intérêt de la programmation dynamique est de permettre de donner (quand c'est possibles) des solutions exactes à ces problèmes en un temps raisonnable (polynomial dans un certain nombre de cas).

Les deux points-clés de l'utilisation de la programmation dynamique sont les suivants :

— Chercher une **équation de récurrence** (exprimer la solution pour une entrée donnée en fonction des solutions d'entrées de taille inférieure)

— **Stocker en mémoire** les solutions des sous-problèmes susceptibles d'être réutilisées plusieurs fois.

2.2 Approches *Bottom-Up* et *Top-Down*

On a vu que la programmation dynamique se basait sur une expression par récurrence de la solution à un problème donné, en fonction des solutions de sous-problèmes.

Cependant, la fonction `fibo` dont le programme est donné ci-dessus est un programme itératif : on calcule les valeurs de $F(n)$ d'abord pour les petites valeurs de n, puis pour les plus grandes valeurs en fonction des valeurs déjà calculées.

En fait, lorsque l'on connaît l'équation de récurrence pour exprimer la solution d'un problème en fonction de celles des sous-problèmes, il y a deux approches possible pour employer la programmation dynamique :

— **De bas en haut (bottom-up)** : On suppose que l'ensemble des instances est muni d'une relation d'ordre bien fondé, de telle sorte que la solution d'une instance s'exprime en fonction des solutions d'instances inférieures.

  On calcul ensuite les solutions des sous-problèmes pour les instances inférieures à celle du problème, par ordre croissant, puis on les stocke dans une structure adéquate : les solutions pour les instances inférieures ont déjà été calculées lorsqu'elles doivent être (ré-)utilisées pour les calculs de solutions d'instances supérieures. On parcourt les sous-problèmes à résoudre de façon itérative.

— **De haut en bas (top-down)** : On utilise la récursivité et l'équation de récurrence pour définir la fonction, mais on utilise une structure pour stocker les solutions des sous-problèmes. Si la solution a déjà été calculée, on la récupère dans la structure ; sinon, on la calcule et on la stocke dans la structure pour qu'elle puisse être réutilisée.

  Le processus consistant à soit récupérer une solution, soit la créer et la stocker dans une structure lors de sa première utilisation est appelée **mémoïsation** (ou `memoization`).

En règle générale, l'approche top-down se fait de la façon suivante :

— on utilise une fonction auxiliaire, qui en plus de la variable correspondant à l'instance prend en entrée une structure (dictionnaire, tableau).
— A la fin de l'appel de la fonction, on modifie le dictionnaire en y ajoutant la solution obtenue pour l'instance en entrée.
— Dans les cas récursifs à l'intérieur du programme, on fait les appels récursifs sur le même dictionnaire (qui contiendra alors les solutions déjà calculées).

On utilise ensuite cette fonction auxiliaire sur l'entrée souhaitée et un dictionnaire initialement vide.

2.3 Retour sur Trace (Backtracking)

On considère un problème consistant à trouver une solution vérifiant une condition parmi un ensemble V de solutions potentielles.

L'exploration exhaustive consiste à tester pour chacune des solutions potentielles dans V si celle-ci est bien solution du problème, jusqu'à en trouver une.

Dans certaines situations, une solution potentielle d'un problème est une combinaison de plusieurs informations, de sorte qu'elle peut s'exprimer sous la forme de la donnée d'une information et d'une solution potentielle à un sous-problème. Par exemple, une grille de Sudoku peut être remplie correctement si, pour une case vide donnée de la grille, parmi les 9 grilles obtenues en remplissant la case par une valeur entre 1 et 9, il en existe une qui puisse être remplie correctement.

Le **retour sur trace** (backtracking) consiste à construire l'ensemble des solutions potentielles à explorer sous la forme d'un arbre. Dès que l'on sait que la construction ne peut pas donner de solution valable, on retourne en arrière et on regarde les autres choix possibles au même niveau.

Exemples :

— L'algorithme de Quine est un algorithme de backtracking.

— Résolution du Sudoku : à chaque case, on envisage les 9 possibilités de remplissage ; on procède de même pour les sous-grilles obtenues jusqu'à arriver sur une grille incohérente.

— Problème des 8 reines : peut-on positionner 8 reines sur un échiquier sans qu'il n'y en ait deux qui soient sur la même ligne, la même colonne ou la même diagonale ?