République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mohamed Boudiaf de M'sila
Faculté de Mathématiques et Informatique
Département d'informatique

INITIATION A L'OPTIMISATION COMBINATOIRE

- SUPPORT DE COURS -

Pour Master 1 Informatique Décisionnelle et Optimisation

Par :

Dr. Allaoua HEMMAK

(MAI 2023)

Préface

Ce document consiste en un support de cours d'optimisation combinatoire destiné aux étudiants de master 1 de la spécialité Informatique Décisionnelle et Optimisation (IDO) dans le cadre de l'enseignement des modules d'optimisation combinatoire. Le cursus de cette spécialité est enseigné au département d'informatique, faculté de mathématiques et informatique, université Mohamed Boudiaf de M'sila depuis septembre 2015. Plus précisément, la matière cible s'intitule 'Optimisation combinatoire 1' (OC1, coefficient 03, 05 crédits) faisant partie de l'unité fondamentale 1 dont l'objet est l'initiation à l'optimisation combinatoire et étude des méthodes de résolution exactes et dont les prérequis sont les bases d'algorithmique, complexité et structures de données.

Cette modeste contribution s'émane de notre conscience de l'importance de l'optimisation combinatoire dans tous ses aspects tant pour les spécialistes du domaine que pour la recherche opérationnelle en général, d'une part, et des difficultés auxquelles font face nos étudiants pour assimiler ses notions, solutionner des problèmes inhérents ou aborder des projets qui font appel à ses concepts. Le terme solutionner désigne ici tout un processus de résolution qui consiste à spécifier, formuler, modéliser puis implémenter la solution adéquate du problème en question. En effet, en partant de notre modeste expérience dans l'enseignement de cette matière, nous nous sommes définis les objectifs permettant aux étudiants de :

i. S'initier à l'optimisation combinatoire ;
ii. Assimiler les méthodes exactes ;
iii. Apprendre à formuler, modéliser, analyser les problèmes d'optimisation combinatoire.
iv. Acquérir des compétences de conception et d'implémentation des solutions aux problèmes ;

Dans ce contexte, nous avons opté de structurer ce document comme suit :

- Un chapitre premier qui étale un rappel de la notion d'algorithme, ses types, ses propriétés et ses paradigmes, du moment qu'il s'agit de l'élément clé dans la résolution des problèmes d'optimisation.

- Dans un deuxième chapitre, on abordera la complexité des algorithmes, ses notations et méthodes de calcul, puisqu'elle constitue l'outil primordial permettant d'évaluer, analyser et comparer les algorithmes.
- Le troisième chapitre sera consacré aux concepts de base de l'optimisation combinatoire et abordera une gamme variée d'exemples de problèmes afin de mieux expliciter les notions présentées.
- Les quatrième et cinquième chapitres seront dédiés respectivement aux deux méthodes exactes les plus utilisées en optimisation en l'occurrence : la programmation dynamique et la méthode par séparation et évaluation avec des exemples illustratifs appropriés.
- Enfin, vu l'utilité du paradigme glouton, le dernier chapitre en présente le principe et des exemples d'applications.

Dans chaque chapitre, des exemples appropriés sont inclus afin de permettre aux étudiants de mieux digérer les concepts présentés. De plus, une série d'exercices non résolus est insérée à la fin de chaque chapitre pour permettre aux étudiants de s'entrainer, s'autoévaluer et bien assimiler les notions du cours.

Dr. Hemmak Allaoua (allaoua.hemmak@univ-msila.dz) mai 2023

Table des matières

CHAPITRE I : RAPPELS D'ALGORITHMIQUE

Plan

1. Définitions
2. Exemple d'algorithmes
3. Propriétés d'un algorithme
4. Types d'algorithmes
5. Paradigmes d'algorithmique
6. Exercices

1. Definitions

(Il y en a plusieurs)

Un algorithme peut être défini comme :

- Une suite d'opérations (d'actions, d'ordres, d'instructions) …;
- Des structures de contrôle (SDC) manipulant des structures de données (SDD) …;
- Un ensemble d'objets ayant des attributs et des opérations échangeant des messages …;
- D'une manière générale : une représentation automatisable d'une méthode …;

… dont l'implémentation permet de résoudre un certain problème par une machine.

SDC

a) Affectation :
   - Variable = Expression ;
   - Lecture (scanf);
   - Ecriture (prinf);

b) Condition :
   - Simple (if simple);
   - avec alternative (if avec else, elseif);
   - multi-choix (switch);

c) Itération :
   - Un nombre connu d'itérations (for) ;
   - Un nombre d'itérations inconnu (while) ;
   - Au moins une itération (do … loop until) ;

d) Récursion (self appel) :
   - Simple : un seul self appel ;
   - Multiple : plusieurs self appels ;
   - Mutuelle : appels mutuels ;
   - Terminale : l'appel récursif est la dernière action ;

Remarque

Toutes ces structures pourraient être en séquence et/ou en parallèle.

SDD

- Concrètes (physiques)
  - → Constante, variable simple (bit, boolean, byte, char, int, float , double, string, …) ;
  - → Tableau (uni ou multi dimensionnel) ;
  - → Liste chainée (à l'aide de pointeurs) ;
- Abstraites (implémentées par des SDD concrètes) :
  - → Linéaires (Ensemble, Liste, Pile, File) ;
  - → Arborescentes (Arbre, Graphe) ;

2. Exemple d'algorithmes

Calcul du PGCD de deux entiers

```
Int gcd (int a, int b)
    {  while (a<>b)
        if (a>b) a-=b ;
        else b-=a ;
     Return   a}
```

Recherche d'un entier dans un tableau d'entiers

```
Recherche (int[] A, int n, int x)
    {  for (i=0 ; i<n ; i++)
        If (A[i]==X) return i ;
     Return   false}
```

Calcul du minimum d'un tableau d'entiers

```
Min (int[] A, int n, )
    {  M = A[0] ; ind = 0 ;
for (i=0 ; i<n ; i++)
        If (A[i]<M) { M = A[i] ; ind = i ;} ;
     Return   ind}
```

Tri d'un tableau d'entiers par insertion

```
tri_insertion(tableau A, n)
    {  for (i=1 ; i<n ; i++)
        {x = A[i]
        j = i
        while (j > 0 and A[j - 1] > x)
              {  T[j] = A[j - 1] ;
               j - - ;}
        T[j] ← x } }
```

Tri d'un tableau d'entiers par sélection

```
tri_selection(tableau A, n)
    {  for (i=1 ; i<n-1 ; i++)
       { min ← i  ;
        for (j=i+1 ; i<n ; j++)
           if(A[j] < A[min]) min = j ;
        if (min != i) échanger (A[i] et A[min]) }}
```

Tri d'un tableau d'entiers par bulles

```
triBulles (tableau A, n)
{
for (i=0 ; i < n-1; i++)
   for (j=0 ; j < n-i-1; j++)
     if (A[j] > A[j+1])
           échanger  A[j] et  A[j+1] ;
}
```

Quiz : pour chacun des algorithmes ci-dessus, calculer la taille de l'espace mémoire requis et le nombre d'opérations.

3. Propriétés d'un algorithme

Pour qu'il soit valide, un algorithme doit satisfaire trois propriétés :

- Terminaison (il faut qu'il se termine) ;
- Correction (il faut qu'il soit conforme à la spécification du problème à résoudre) ;
- Complétude (doit solutionner toutes les instances du problème en question) ;

Sinon, l'algorithme n'est pas valide.

4. Types d'algorithmes

(Il y en a plusieurs classifications)

a) Selon le mode d'exécution :

- Séquentiel : ses actions s'exécutent séquentiellement (l'une après l'autre) ;
- Parallèle : certaines de ses actions s'exécutent en parallèle (en même temps sur un ordinateur multiprocesseur ou avec un seul processeur mais multicœurs) ;
- Distribué : certaines de ses actions s'exécutent en parallèle (en même temps sur un réseau d'ordinateurs) ;

b) Selon la qualité de la solution fournie :

- Exact (optimal) : il fournit une solution exacte (prouvée formellement) qui répond exactement au problème.

Exemples

i. L'algorithme de résolution d'une équation du second degré par la méthode du discriminant) ;
ii. Le tri par bulles ;
iii. L'algorithme de Bellman calculant le plus court chemin dans un graphe.

- Approché : il fournit une solution approchée (prouvée empiriquement) qui répond approximativement au problème.

Exemples

i. L'algorithme de résolution d'une équation du troisième degré par le théorème des valeurs intermédiaires) ;

ii. L'algorithme de calcul d'une valeur approchée de $\pi, e^x, \sqrt{x}, \ldots$ par le développement limité de Taylor ou autre méthode ;

c) Selon le déterminisme de ses actions :
- Déterministe : pas d'actions probabilistes (soumises au hasard).
- Indéterministe (stochastique) : l'algorithme comporte des actions probabilistes (soumises au Hazard).

Exemple

Algorithme d'un jeu, if (rand(0,1) < 0.5) prinf ('face') else prinf ('pile') ;

Quiz : implémenter un petit jeu de dé.

Remarque

Tous les jeux sont des algorithmes stochastiques.

d) Selon le mode des actions répétitives :
- Itératif : l'algorithme n'appelle (n'utilise) pas lui-même (directement ou indirectement).
- Récursif : l'algorithme appelle lui-même (directement ou indirectement).
  On distingue : la récursivité :
  - Simple : un seul appel récursif : calcul du factoriel d'un entier.
    int fact(int n) { if(n==0 or n==1) return 1 ; else return n*fact(n-1);}
  - Multiple : plusieurs appels récursifs : tri fusion, calcul du nième terme de la suite de Fibonacci (récursivité double).
    int Fibo(int n) { if(n==0 or n==1) return n ; return Fibo(n-1)+Fibo(n-2);}
  - Mutuelle : des appels mutuels : calcul de la liste des nombres premiers inférieurs à n.
    Bool Prem(int n) {for each i in ListePrem(sqrt(n)) if (n % i ==0) return 0; return 1;}
    Int[] ListPrem(int n){ l=null; for (j=2; j<=n ; j++) if (Prem(j)) l.append(j);}

Remarques
- ✓ Elaborer un programme récursif est beaucoup plus aisé qu'élaborer un programme itératif équivalent.
- ✓ On peut toujours passer d'un programme itératif à un programme récursif équivalent et vice-versa. Cette dernière opération est dite élimination de la récursivité.
- ✓ Sachant que tout programme, pour être exécuté, il doit être traduit en langage machine et puisque, dans tous les systèmes d'exploitation et toutes les architectures hardware, ce dernier n'est jamais récursif, il est donc toujours recommandé d'éviter la récursivité. Au fait, c'est le compilateur qui se charge d'éliminer la récursivité à votre place. Ainsi, éviter la récursivité veut dire éviter d'éventuelles erreurs sémantiques très difficile à détecter dans certains cas.
- ✓ En effet, si on opte pour un programme récursif, il faut toujours mettre l'appel récursif comme dernière action. Dans ce cas on dit que la récursivité est terminale.

Exemples

i. Récursivité non terminale :
   int fact(int n) { if(n==0 or n==1) return 1 ; else return fact(n-1)*n;}
ii. Récursivité terminale :
   int fact(int n) { if(n==0 or n==1) return 1 ; else return n*fact(n-1);}

e) Selon le mode d'introduction des données :
   - On line : les données ne sont connues qu'au moment de l'exécution de l'algorithme.
   - Off line : les données sont disoinibles avant l'exécution de l'algorithme.

f) Selon la complexité : (voir chapitre II)
   - Polynomial : le nombre de ses opérations est une fonction polynomiale de la taille de son input.
   - Exponentiel : le nombre de ses opérations est une fonction exponentielle de la taille de son input.
   - Pseudo-polynomial : le nombre de ses opérations est une fonction polynomiale valeurs de son input.

5. Paradigmes d'algorithmique

Ce sont les stratégies, styles de conception des algorithmes pour lesquels on opte lors du design d'un algorithme. Ce choix dépend de plusieurs facteurs : la complexité du problème, la taille de son input, l'environnement à utiliser pour l'implémentation, …

- Force brute : on utilise la définition littéralement.
  Comme utiliser la formule : $C_n^k = \frac{n!}{k!(n-k)!}$ pour calculer C_n^k.
- Exhaustif : énumérer tous les cas possibles ;

Exemples

i. Pour trier un tableau, on teste toutes les permutations possibles des éléments du tableau ;
ii. Pour calculer le plus court chemin entre a et b, on explore tous les chemins existants entre a et b.

- Diviser pour régner : diviser le problème en sous-problèmes de tailles inferieures, résoudre les sous problèmes (en général, récursivement) puis fusionner leurs solutions pour dégager la solution du problème entier.

Exemples

i. Recherche dichotomique ;
ii. Tri fusion ;
iii. Utiliser la formule : $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ pour calculer récursivement C_n^k.
iv. Calcul du nième terme de la suite de Fibonacci récursivement.

- Programmation dynamique : un cas particulier du 'diviser pour régner' où les sous problèmes sont imbriqués dont les solutions sont stockées dans une table afin d'éviter de résoudre le même sous problème plusieurs fois (voir plus de détails au chapitre IV).

Exemples

i. Utiliser la formule : $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ pour calculer C_n^k itérativement en utilisant une table contenant les valeurs de C_i^j pour i = 0, n et j = 0, k.
ii. Calcul du nième terme de la suite de Fibonacci itérativement en utilisant une table contenant les valeurs de Fib(i) pour i=0,n.

- Méthode par séparation et évaluation : un cas particulier du 'diviser pour régner' où les sous problèmes sont disjoints formant une arborescence où on doit encadrer la solution d'un sous problème par deux bornes inférieure et supérieure, un bon choix de ces derniers permettra d'abandonner plusieurs branches afin d'éviter l'exploration exhaustive et accélérer l'algorithme (voir plus de détails au chapitre V).

Exemples

Pour calculer le plus court cycle hamiltonien en partant du sommet a, on teste tous les successeurs de a et on estime pour chaque cas la longueur maximale (borne supérieure) et minimale (borne inférieure) du cycle ; trois situations sont alors possibles :

i. Si les deux bornes sont égales, la solution optimale est trouvée et on arrête l'exploration.
ii. Si la borne inférieure est strictement supérieure à la borne supérieure, on abandonne cette branche puisqu'elle non réalisable.
iii. Si la borne inférieure est strictement inférieure à la borne supérieure, on refait le même procédé avec les successeurs de ce nœud (on explore la branche.)

Remarque

Cette méthode s'appuie sur les heuristiques de calcul des deux bornes ; plus les deux bornes sont proches, plus l'algorithme converge rapidement et vice-versa.

- Algorithme glouton : lorsque la solution est construite par étapes (ce qui est la plupart des cas), la meilleure décision (optimale) est prise à chaque étape localement sans tenir compte ni du passé ni du futur du processus de décisions dans l'espoir d'aboutir à une bonne solution (voir plus de détails au chapitre VI).

Exemples

i. Pour calculer le plus court chemin de a à b, on prend le sommet le plus proche de a, soit c, puis le plus proche de c, soit d, puis le proche de d et ainsi de suite sans tenir compte de la longueur cumulée par ces choix.
ii. Pour rendre le minimum de pièces de monnaie, on commence par la pièce de plus grande valeur possible et ainsi de suite.

Quiz : écrire les algorithmes correspondants à ces deux exemples.

Remarque

L'algorithme glouton pourrait fournir une solution exacte (optimale) dans certains cas (voir plus de détails au chapitre IV.).

6. Exercices

Exercice 1

Ecrire trois algorithmes calculant le nième terme de la suite de Fibonacci.

Exercice 2

Ecrire un algorithme qui teste si un tableau est trié.
Ecrire un algorithme qui calcule le max et le min d'un tableau d'entiers.
Ecrire un algorithme qui calcule les 2 plus grands éléments d'un tableau d'entiers.
Ecrire un algorithme qui calcule le nombre de pics dans un tableau d'entiers.

Exercice 3

Ecrire un algorithme qui teste si un entier est premier.

Exercice 4

Ecrire un algorithme qui teste si un réel x est une racine d'un polynôme P de degré n.
Ecrire un algorithme qui calcule la somme de 2 polynômes de degré n.
Ecrire un algorithme qui calcule le produit de 2 polynômes de degré n.
Ecrire un algorithme qui calcule la dérivée première d'un polynôme de degré n.

Exercice 5

Ecrire un algorithme qui calcule le graphe symétrique d'un graphe orienté.
Ecrire un algorithme qui teste si un graphe orienté est complet.
Ecrire un algorithme qui imprime les sommets d'un graphe en le parcourant en profondeur.
Ecrire un algorithme qui imprime les sommets d'un graphe en le parcourant en largeur.
Note. Supposer que le graphe est implémenté par sa matrice d'adjacence.

Exercice 6

Ecrire un algorithme qui teste si un arbre binaire est complet.
Ecrire un algorithme qui imprime les sommets d'un arbre binaire en le parcourant en profondeur.
Ecrire un algorithme qui imprime les sommets d'un arbre binaire en le parcourant en largeur.

Exercice 7

Implémenter un jeu qui cache un entier compris entre 1 et 1000 et le joueur le devine.

Exercice 8

Ecrire un algorithme qui résout l'équation $ax^2+bx+c=0$.
Ecrire un algorithme qui résout l'équation $ax^3+ bx^2+cx+d=0$ par une méthode approchée itérative qui utilise le théorème des valeurs intermédiaires sur l'intervalle [p,q].
Ecrire un algorithme qui résout un système de 2 équations linéaires à deux variables réelles.

Exercice 9

Ecrire un algorithme qui calcule la somme de deux matrices.

Ecrire un algorithme qui calcule le produit de deux matrices.
Ecrire un algorithme qui calcule le produit d'une matrice par un vecteur.
Ecrire un algorithme qui calcule le déterminant d'une matrice carrée.

Exercice 10

Soit l'algorithme suivant :

```
Int p (int a ; int b){
While (a<>b) if (a>b) a-=b ; else b-=a ;
Return a; }
```

Que fait cet algorithme ?
Calculer en fonction n le nombre de ces opérations.
Ecrire un algorithme qui teste si un nombre est premier.
Ecrire un algorithme qui teste si deux entiers sont premiers entre eux.
Ecrire un algorithme qui calcule les nombres premiers inférieurs à n.
Ecrire un algorithme qui recherche les entiers a,b,c inférieurs à n et satisfont : $a^2+b^2=c^2$.

Exercice 11

Ecrire un algorithme qui teste si un réel x est une racine d'un polynôme P de degré n.
Ecrire un algorithme qui calcule la somme de 2 polynômes de degré n.
Ecrire un algorithme qui calcule le produit de 2 polynômes de degré n.
Ecrire un algorithme qui calcule la dérivée première d'un polynôme de degré n.

Exercice 12

Ecrire un algorithme qui calcule la moyenne, la variance, la médiane et le mode d'une série statistique discrète pondérée (x_i, f_i) ; i =1,n.
Note.
Mode : Le mode est la valeur la plus fréquente dans un échantillon.
Médiane : la médiane est un nombre qui divise en 2 parties la population telle que chaque partie contient le même nombre de valeurs.
Moyenne : La moyenne arithmétique est la somme des valeurs de la variable divisée par le nombre d'individus.
La variance : La variance est la moyenne des carrés des écarts à la moyenne.

Exercice 13

Ecrire un algorithme qui calcule une valeur approchée de $\sqrt{x}$, π , e^x , $\sin x$ en utilisant le développement limité de Taylor.

Exercice 14

Soit l'algorithme suivant :

```
Int Test (int n ; int A[]){
For (i=0;i<n-1;i++) if (A[i]> A[i+1]) return false ;
Return true; }
```

Que fait cet algorithme ?
Calculer en fonction n le nombre de ces opérations.

Ecrire un algorithme qui fusionne deux tableaux d'entiers triés en un seul trié.
Ecrire un algorithme qui calcule les deux plus grands éléments d'un tableau d'entiers.
Ecrire un algorithme qui calcule le nombre de pics dans un tableau.

CHAPITRE II : COMPLEXITE DES ALGORITHMES / DES PROBLEMES

Plan

1. Introduction
2. Définition
3. Calcul de la complexité
4. Notation de Landau
5. Classes de problèmes
6. Exercices

1. Introduction

On a vu dans le chapitre précèdent qu'un programme est de structures de contrôle (les instructions) manipulant des structures de données (variables et constantes).
Pour exécuter votre programme, vous avez donc besoin d'espace mémoire pour stocker les SDD et un processeur pour exécuter les SDC.
L'ordinateur dispose de deux ressources nécessaires pour exécuter un programme : espace mémoire et temps processeur (architecture de Von Neumann RAM-CPU) ;
La complexité d'un algorithme est une mesure de la quantité de ces ressources consommée par l'exécution de cet algorithme ;
On distingue alors :

- → La complexité spatiale : espace mémoire… ;
- → La complexité temporelle : temps CPU… ;

… requis pour l'exécution de cet algorithme.

La première n'importe pas beaucoup avec l'évolution spectaculaire du hardware ;
La seconde, quant à elle, est primordiale et dite alors complexité tout court (par abus de langage) ;
Il est clair que le temps d'exécution d'un algorithme est fonction de ses opérations (plus particulièrement ses opérations fondamentales) et de la taille de son input (la quantité de données que prend l'algorithme en entrée) ;

2. Définition

- ✓ La complexité d'un algorithme est le nombre d'opérations fondamentales que fait cet algorithme en fonction de la taille de son input.
- ✓ L'opération fondamentale est celle qui affecte le plus le temps d'exécution.

Exemples

i. C'est la multiplication dans le calcul de n!.
ii. C'est l'addiction dans le calcul de Fib(n).
iii. C'est la comparaison dans la recherche d'un élément dans un tableau.

- ✓ La complexité est donc une estimation formelle (mathématique) du temps d'exécution de l'algorithme en fonction de la taille de son input.
- ✓ Elle est indépendante de :
  - → De la machine …;
  - → De l'OS …;
  - → Du langage …;

… à utiliser pour exécuter l'algorithme.

Remarque

On distingue, suivant l'instance de données de l'algorithme, la complexité :

- ✓ Dans les pires cas ;
- ✓ Dans les meilleurs cas ;
- ✓ Dans le cas moyen ;

Exemples

Lors de la recherche de votre nome dans une liste de n noms, on fera :

i. Une seule comparaison, dans le meilleur cas, si votre nom est en tête de liste ;
ii. n comparaisons, dans le pire cas, si votre nom est le dernier de la liste ;
iii. $\frac{\sum_{i=1}^{M} r_i}{M}$ comparaisons (M est le nombre d'instances, r_i est le rang de votre nom dans l'instance i) dans le moyen cas.

Remarque

La complexité dans les moyens cas est souvent difficile à évaluer puisque le nombre d'instances est dans la plupart des problèmes infini.

Une définition plus formelle :

La complexité est le nombre d'opérations élémentaires qu'effectue une machine de Turing pour reconnaitre un mot en fonction de la longueur de ce mot.

Elle sert à analyser, évaluer, comparer les algorithmes.

Ce qui importe c'est quand la taille de l'input est assez grande : allure asymptotique du nombre d'opérations.

Exemple

Calcul de x^n

Premier algorithme : en utilisant la définition : $x^n = x*x*\ldots*x$, n fois :

Float power(int n) { p=1 ; for (j=0; j<n-1; j++) p*=x; return p;}

L'opération fondamentale est la multiplication dont le nombre est T(n) = n-1.

On peut en gagner une en introduisant :

Float power(int n) { p=x ; for (j=0; j<n-2; j++) p*=x; return p;} et on aura T(n) = n-2.

Dans les deux cas, quand n est assez grand, T(n) ≈ n (plus précisément T(n) est majoré par c.n),

c.-à-d. : $\lim_{n\to\infty} \frac{T(n)}{n} = 0$.

Deuxième algorithme : en utilisant la formule :

$x^n = x^2*x^2*\ldots x^2$, si n est pair et $x^n = x^2*x^2*\ldots x^2*x$ si n est impair.

On aura T(n) ≈ n/2 (et on pourrait faire mieux).

Quiz : peut-on faire mieux ?

3. Calcul de la complexité

Soit P1 et P2 deux programmes et T(x) le nombre d'opérations fondamentales que fait l'algorithme x:

T(x=a) = 1 (une unité)
T(if (condition) P1) = T(P1)
T(if (condition) P1 else P2) = max (T(P1) , T(P2)
T(for (i=0 ; i<k ; i++) P1) = k*T(P1)
T(while (condition) P1) = max_iterations*T(P1)
T(P1; P2)= T(P1) + T(P2)
T(P1 // P2)= max (T(P1) , T(P2))

Si le programme est itératif et n est la taille de son input : T(n) est une fonction de n ;

Exemples

T(n) = 3n+2 ; T(n)= log n ; T(n)= n^2+n-1 ; T(n)= 2^n ;…

Remarque

Si le programme est récursif et n est la taille de l'input :
T(n) est récurrente et est une fonction de T(n-1) ou T(n/2) ou T(n-2), ….; avec un cas trivial comme T(0)= 1 et/ou T(1)= 1 , ….

Exemples

i. T(1) = 1 et T(n)= T(n-1) + 2 → T(n) = 2n+1 ;
ii. T(1) = 0 et T(n)= T(n-1) + n → T(n) = n(n-1) = n^2-n ;
iii. T(1) = 1 et T(n)= 2 *T(n/2) + 1 → T(n) = $\log_2 n$;

Quiz : prouver ces résultats.

4. Notation de Landau

Le mathématicien allemand Edmund Landau s'est intéressé à l'étude asymptotique des fonctions numériques à variable entière en définissant les notations suivantes:

$f = O(g) \Leftrightarrow \exists n_0 \in N , \exists c \in R , \forall n \geq n_0 : f(n) \leq c.g(n)$
$f = \Omega(g) \Leftrightarrow \exists n_0 \in N , \exists c \in R , \forall n \geq n_0 : f(n) \geq c.g(n)$.
$f = \Theta(g) \Leftrightarrow \exists n_0 \in N , \exists c1,c2 \in R , \forall n \geq n_0 : c1.g(n) \leq f(n) \leq c2.g(n)$.

La notation la plus utilisée est O (se lit big o). On écrit : f = O(g(n)) ou f ∈ O(g(n)) et on dit que l'algorithme est en O(g(n)). Ceci veut dire : Si n est assez grand, f(n) est de l'ordre de g(n) ;
Ces notations permettent de décrire l'allure la courbe représentative de la fonction f quand n est assez grand ($n \to \infty$).

Exemples

Si T(n) = 3n+2 alors T(n)= O(n) puisque on peut aisément trouver un réel c et un entier n_0 tels que T(n) ≤ c.n pour tout n ≥ n0.

En effet : T(n) ≤ c.n ⇔ 3n+2 ≤ c.n ⇔ $n \geq \frac{2}{c-3}$

En prenant par exemple c = 4 et n_0=2 on trouve que $\exists n_0 \in N$, $\exists c \in R$, $\forall n \geq n_0$: T(n) ≤ c.n

c.-à-d. T(n)= O(n).

De la même façon on peut démontrer que :

$3n+2 = \Omega(n)$

$3n+2 = O(n^2)$

$2n^2+3n+1 = O(n^2)$

$2n^2+3n+1 = O(n^3)$

$2n^2+3n+1 = \Theta(n^2)$

$2n^2+3n+1 = \Omega(n^2)$

$2n^2+3n+1 = \Omega(n)$

$\log n - 3 = O(\log n)$

Remarques

- $f = O(g) \Leftrightarrow g = \Omega(f)$
- $f = \Theta(g) \Leftrightarrow f = O(g)$ and $g = O(f)$
- Θ est symétrique: $f = \Theta(g) \Leftrightarrow g = \Theta(f)$
- O, Ω sont antisymétriques.
- O, Θ, Ω sont réflexives et transitives.

Quiz : démontrer ces propriétés.

Complexités usuelles

T(n)	Notation en O	Dénomination
Constante (1, 10, 500,,..)	O(1)	Constante (Polynomiale)
an+b	O(n)	Linéaire (Polynomiale)
aLog n+b	O(logn)	Logarithmique (Polynomiale)
an^2+bn+c	$O(n^2)$	Quadratique (Polynomiale)
anlogn+b	O(nlogn)	Quasi logarithmique (Polynomiale)
Polynôme en n degré k	$O(n^k)$	Polynomiale
$c.a^n$ (a >1)	$O(a^n) \approx O(2^n)$	Exponentielle
c.n !	$O(n!) \approx O(2^n)$	Exponentielle

(a, b, c des constantes)

Exemples

Comparer les algorithmes de tri de nombres

i. **Enumération (**idiot) nombre d'opérations borné par c.n!
ii. **Sélection (**bête) nombre d'opérations borné par $a.n.(n-1)/2 = O(n^2)$
iii. **Rapide (**malin) nombre d'opérations borné par a.n.logn = O(nlogn)

Si une opération se déroule en 10^{-6}seconde, le nombre d'opérations est de l'ordre de :

n	Log n	n	n^2	2^n	n!
10	2.3×10^{-6} s	10^{-5} s	10^{-4} s	10^{-3} s	3.6 s
20	3.2×10^{-6} s	2×10^{-5} s	4×10^{-4} s	33.55 s	491×10^{9} a
50	3.9×10^{-6} s	5×10^{-5} s	25×10^{-4} s	35.7 a	-
	Complexités polynomiales			Complexités exponentielles	

La figure II.1 ci-dessous illustre les allures graphiques des différentes complexités.

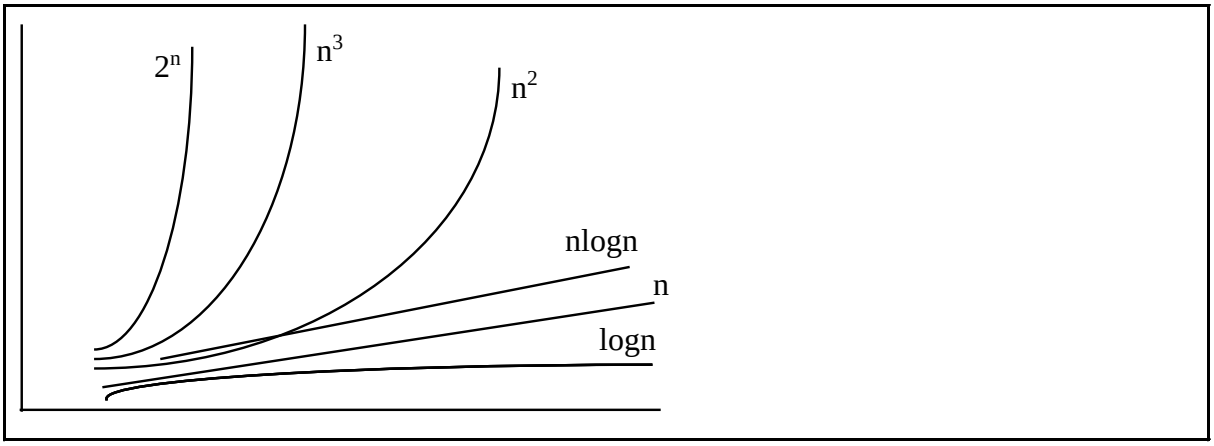


Figure II. 1. Allure de quelques courbes.

5. Classes de problèmes

Problème de décision

Un problème de décision est un problème dont la solution est 'vrai' ou 'faux' (oui ou non).

Exemples

i. L'élément x est-il dans le tableau A[] ?
ii. Le tableau A[] est-il trié par ordre croissant ?
iii. x est-il le max dans le tableau A[] ?
iv. n est-il premier ? Le problème de primalité (Prime Problem, PP).
v. Peut-on colorer les sommets d'un graphe par 3 couleurs tels que deux sommets adjacents aient 2 couleurs différentes ? Problème de k-Coloration (k-Coloring Problem).
vi. Existe-il un cycle hamiltonien de longueur *l* dans le graphe G ?
vii. Peut-on placer 8 reines sur un échiquier sans qu'elles se menacent ?

Définitions

✓ La complexité d'un problème est celle du meilleur algorithme le résolvant.

Meilleur = le plus rapide (le plus efficace).

Classification des problèmes de décision : P , NP , NP-complet , NP-difficile.

Approche de la théorie de la complexité, *Théorie développée à la fin du 20ème siècle (S. Cook 1970 et L. Levin 1973).*

Classes P

On dit qu'un problème de décision est dans la classe P
(Polynomial time) ou « facile easy » si et seulement s'il existe un algorithme polynomial déterministe le résolvant.

Exemples

i. L'élément x est-il dans le tableau A[] ?
ii. Le tableau A[] est-il tiré par ordre croissant ?
iii. Existe-il un cycle eulérien de longueur minimale dans le graphe G ?

Classes NP

On dit qu'un problème est dans la classe NP (Non deterministic Polynomial time) si et seulement s'il existe un algorithme polynomial non déterministe le résolvant , c.-à-d., connaissant une solution, elle peut être vérifiée par un algorithme polynomial déterministe.
Le seul algorithme déterministe pouvant résoudre ce problème est exponentiel.

Exemples

i. Peut-on colorer les sommets d'un graphe par 3 couleurs tels que deux sommets adjacents aient 2 couleurs différentes ?
ii. Existe-il un cycle hamiltonien de longueur minimale dans le graphe G ?
iii. Peut-on placer 8 reines sur un échiquier sans qu'elles se menacent ?

Quiz : discuter le nombre de situations possibles.

Réduction polynomiale

On que le problème A est une réduction polynomiale du problème B (ou A se réduit polynomialement à B) si et seulement si A peut être résolu en un temps polynomial en fonction du nombre d'appels de B et on écrit : $A \leq_P B$.

Problème NP-difficile

Un problème X est dit NP-difficile si et seulement si tout problème de NP peut être réduit polynomialement à X.
$A \in$NP-H SSI $\exists B \in NP$ $A \leq_P B$ (NP-Difficile en anglaise NP-Hard ou NP-H)
(NP-H) $\cap$ NP = NP-C et NP-C $\subseteq$ NP Il est clair que P $\subseteq$ NP mais est-ce-que NP$\subseteq$P?
P=NP ? Problème ouvert millénaire 1.000.000 dollars !

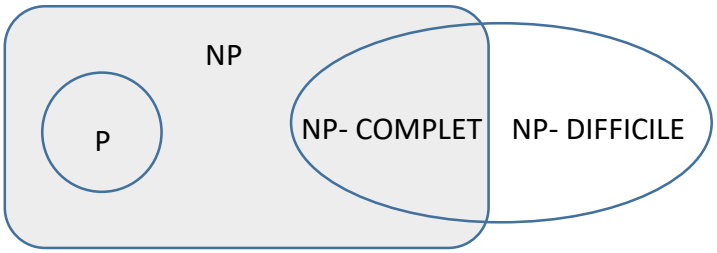


Figure II. 2. Classes de problèmes.

Pour montrer qu'un problème A est polynomial, il faut trouver un algorithme pour le résoudre et prouver que cet algorithme s'exécute en un temps qui augmente de façon polynomiale en fonction de la taille de l'instance traitée.

Pour montrer qu'un problème A est NP-complet, on choisit un problème déjà connu pour être NP-complet, soit A_{nc}, et on montre que A_{nc} peut se "transformer" en A (figure 3).

Donc, si on savait résoudre A, on saurait résoudre A_{nc} (A est aussi difficile que A_{nc})
Or, on ne sait pas résoudre P_{nc} : donc il va sûrement être difficile de résoudre P.
A va, à son tour, être classé NP-complet.

Si on savait résoudre facilement A on saurait résoudre aussi A_{nc};
Or on ne sait pas résoudre A_{nc}, A est donc sûrement difficile à résoudre.

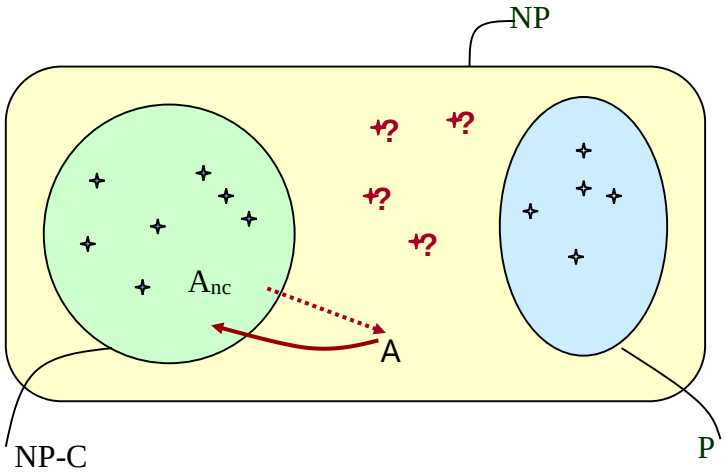


Figure II. 3. Représentation d'une réduction polynomiale.

Le problème SAT ("satisfiabilité" d'une expression logique)
Etant donne une fonction booléenne f à n variables booléennes xi, i=1,n.
Existe-t-il un vecteur x de $\{0,1\}^n$ pour lequel f(x)=1 (true)?

Exemple
$f(x,y,z,t) = (x \vee y \vee z) \wedge (x \vee y \vee \bar{t}) \wedge (y \vee t) \wedge (\bar{x} \vee z)$
Peut-on affecter des valeurs vrai ou faux aux variables de telle façon que l'expression soit vraie ?
Une solution : x=vrai, y=faux, z=vrai , t=vrai.

Quiz : discuter le nombre de combinaisons possibles.

Le théorème de Cook

SAT est le premier problème NP-complet connu.

Stephen Cook a classé le problème SAT comme étant NP-complet.

Nous y reviendrons plus en détails dans le chapitre III.

6. Exercices

Exercice 1

Exprimer les fonctions suivantes en notation O , Ω puis θ :

Considérons un tableau de n entiers.

1) Ecrire l'algorithme de tri par bulles.
2) Ecrire l'algorithme de tri par fusion.
3) Calculer les complexités des deux algorithmes.+

Exercice 2

Pour chacun des fonctions $T_i(n)$ suivant, déterminer sa complexité asymptotique dans la notation Grand-O.

T0(n) = 3n = O(n)
T1(n) = 6n3 + 10n2 + 5n + 2 = O(n3)
T2(n) = 3log2 n + 4 = O(logn)
T3(n) = 2n + 6n2 + 7n = O(2n)
T4(n) = 7k + 2 = O(1)
T4(n) = 4log2 n + n = O(logn)
T5(n) = 2log10 k + kn2 = O(n2)

Exercice 3

Considérer les deux algorithmes A1 et A2 avec leurs temps d'exécution $T_1(n) = 9n^2$ et $T_2(n) = 100n$ respectivement.

1. Déterminer la complexité asymptotique des deux algorithmes dans la notation Grand-O. Quel algorithme a la meilleure complexité asymptotique ?
2. Montrer que vos solutions sont correctes en spécifiant un c et un n_0 par algorithme afin que la relation suivante soit satisfaite : $O(f) = \{g | \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq cf(n)\}$
3. Calculer les temps maximaux d'exécution des deux algorithmes Ti(n) pour n = 1, 3, 5, 10, 14.
4. Ebaucher les graphes des deux fonctions Ti dans un même système de coordonnés (abscisse n, ordonné Ti(n)).
5. Etudier quel algorithme est le plus efficace en fonction de n ?
6. Quelle est la complexité asymptotique de l'algorithme suivant ? Quelle règle avez-vous appliqué ?

```
Début
    appeler A1
    appeler A2
Fin
```

Exercice 4

Addition de matrices

Considérer les deux matrices quadratiques A et B de taille n :

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}, B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix},$$

L'addition de ces deux matrices donne la matrice C quadratique de taille *n*:

$$C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}$$ Avec $cij = aij + bij \; \forall i, \forall j$

1. Définir le type des matrices quadratiques et déclarer les variables A,B, et C.
2. Ecrire un algorithme qui effectue l'addition des deux matrices A et B et stocke les résultats en C.
3. Déterminer la fonction de temps maximale ("worst case") T(n) pour des matrices de taille *n*.
4. Déterminer la complexité Grand-O pour des matrices de taille n.

Exercice 5

Multiplication de matrices

La multiplication des deux matrices quadratiques de taille n donne la matrice C quadratique de taille n:

$$C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}$$ avec $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \quad \forall i, \forall j$

1. Ecrire un algorithme qui effectue la multiplication des deux matrices A et B et stocke les résultats en C.
2. Déterminer la fonction de temps maximale ("worst case") T(n) pour des matrices de taille n.
3. Déterminer la complexité O(n) pour des matrices de taille *n*.

Exercice 6

Écrire l'algorithme qui recherche un élément dans un vecteur de taille n. Calculer la complexité temporelle en fonction du nombre de comparaisons dans le pire et dans le meilleur des cas. Refaire les calculs en fonction du nombre d'accès au vecteur.

Exercice 7

Écrire l'algorithme de tri par sélection. Calculer la complexité temporelle en fonction de nombre de comparaisons et de permutations dans le pire et dans le meilleur des cas. Refaire les calculs en fonction du nombre d'accès au vecteur.

Exercice 8

On considère deux manières de représenter ce que l'on appelle des « matrices creuses », c'est-à-dire des matrices d'entiers contenant environ 90% d'éléments nuls :

a) La matrice est représentée par un tableau à deux dimensions dont les cases contiennent les éléments.
b) La matrice est représentée par un tableau à une dimension. On ne s'intéresse qu'aux éléments de la matrice *qui ne sont pas nuls*. Chaque case du tableau contient un triplet (i, j, a) correspondant à l'indice de ligne, l'indice de colonne, et la valeur d'un élément non nul.

Le problème considéré consiste à calculer la somme des éléments d'une matrice. On demande d'écrire un algorithme permettant de calculer cette somme, pour chacune des deux représentations, puis de comparer leur complexité spatiale (espace mémoire occupé) et leur complexité temporelle (nombre d'opérations à effectuer). Que peut-on conclure de cette comparaison ? Montrer qu'il

existe une valeur critique du nombre d'éléments non nuls à partir de laquelle une méthode l'emporte sur l'autre.

Exercice 9

Pour chacun des algorithmes suivants évaluer le nombre d'opérations :

```
Algo 1
        Pour i allant de 1 à n faire
                Afficher(xi)


Algo 2
        Pour i allant de 1 à n faire
                Pour j allant de 1 à n faire
                        afficher(xi+xj)

Algo 3
    Pour i allant de 1 à n faire
      Pour j allant de 1 à n faire
        Pour k allant de 1 à n faire
                Pour l allant de 1 à n faire
                  Pour m allant de 1 à n faire
                        afficher(xi+xj+xk+xl+xm)
```

Exercice 10

On considère, pour effectuer la recherche d'un élément dans un tableau, la recherche séquentielle et la recherche dichotomique. On s'intéresse à leur complexité temporelle.

Pour cela, considérer un tableau ayant mille éléments (version triée, et version non triée). Pour chaque algorithme, et pour chaque version du tableau, combien de comparaisons sont à effectuer pour :

- Trouver un élément qui y figure ?
- Trouver un élément qui n'y figure pas ?

Quels sont les cas où le tableau est parcouru complètement et les cas où un parcours partiel est suffisant ?

Conclure en donnant la complexité temporelle pour chaque algorithme

Exercice 11

Soient les 3 algorithmes suivants permettant de calculer la valeur d'un polynôme $p(x) = a0+a1x+a2x^2+...+anx^n$ en un point x. Comparer leurs complexités.

```
Algo 1
        p = a[0] ;
        pour i := 1  à  n  faire
                calculer p+ = a[i]* x^i ;
```

Algo 2

```
        p = a[0] ;
        q = 1;
        pour i = 1  à  n  faire
                q = q * x ;
                p := p  +  a[i] * q ;
```

Algo 3

```
p = a[n] ;
pour i := n à 1, pas -1, faire
        p := p*x + a[i-1] ;
```

Exercice 12

Compléter le tableau ci-dessous

Nombre d'opérations en fonction de la taille n des données	**Avec un ordinateur X**		**Avec un ordinateur Y 100 fois plus rapide que X**	
	Taille maximale (n max) des problèmes traités en 1h	Nombre d'opérations effectuées en 1h	Taille maximale (n max) des problèmes traités en 1h	Nombre d'opérations effectuées en 1h
5n				
Log n				
n^2				
2^n				

Exercice 13

Soit l'algorithme suivant qui effectue la recherche dichotomique du rang (place) d'un nombre A dans une suite triée (par ordre croissant) de n nombres mis dans un tableau à une dimension (vecteur L[i]; i=1,..,n). Cet algorithme fonctionne sous l'hypothèse que A est présent dans la liste.

Algorithme :

```
début
        place = 1 ;
        f = n ;
 tant que  place < f  faire   milieu = (place + f) / 2 ;   si
L[milieu] < A  alors    place = milieu + 1 ;
                sinon f = milieu ;
                finsi;   fait;
fin;
```

Donner la complexité de cet algorithme.

Exercice 14

Écrire l'algorithme de tri à Bulles. Calculer la complexité temporelle en fonction de nombre de comparaisons et de permutations dans le pire et dans le meilleur des cas. Refaire les calculs en fonction du nombre d'accès au vecteur. Comparer avec le tri par sélection.

Exercice 15

Calculer la complexité temporelle de l'algorithme de recherche dichotomique en fonction du nombre de comparaisons dans le pire et dans le meilleur des cas. Refaire les calculs en fonction du nombre d'accès au vecteur. Comparer avec l'algorithme de recherche séquentiel.

Exercice 16

On considère trois tris élémentaires : le tri sélection, le tri par insertion (trois variantes), et le tri bulle. On considère pour un tableau les deux cas extrêmes où le tableau est déjà trié (dans l'ordre croissant), et celui où il est trié dans l'ordre décroissant. Décrire avec précision le comportement et la complexité de chacun des algorithmes dans ces deux cas. Quelles conséquences peut-on en tirer ?

Exercice 17

On considère un tableau à une dimension contenant des lettres majuscules. On désire compter la fréquence de chacune des 26 lettres de l'alphabet. Ecrire deux procédures qui donnent en sortie un tableau de fréquence: l'une où le tableau est parcouru 26 fois, et l'autre (plus performante !) où le calcul est fait en un seul parcours. On pourra supposer que l'on dispose d'une fonction auxiliaire *position(lettre)* qui pour chaque lettre donne sa position dans l'alphabet : *position('A') = 1, ..., position('Z) = 26.*

Exercice 18

L'algorithme ci-contre est celui d'une fonction qui prend en entrée un entier non nul et fournit en sorite un entier non nul.

```
int f(int n) {
if (n == 1) return 1 ;
if (n == 2) return 2 ;
for (int k = f(n - 1) ; k > 1 ; k = f(k - 1))
        if ((n % k) == 0) return f(n - 1) ;
return n ;
}
```

1) Donner les résultats respectifs retournés par cette fonction pour les valeurs 3 , 4 et 5 de n.
2) Construire l'arbre des appels récursifs pour n=5.
3) Expliquer ce que fait cet algorithme. Quel inconvénient présente-t-il ? Quelle solution proposez-vous ?
4) Donner une équation récurrente exprimant le nombre T(n) d'opérations modulo (%) qu'effectue cet algorithme en fonction de n.
5) Donner une version itérative de cet algorithme en $O(n\sqrt{n})$.

Exercice 19

L'algorithme suivant prend en entrée un tableau de n entiers.

1. Expliquer brièvement ce que fait cet algorithme puis évaluer sa complexité en fonction de n dans les pires cas de son input.
2. Ecrire un algorithme BETTER-ALGO-X(A) en O(nlogn) équivalent à ALGO-X(A).
3. Déduire un algorithme ALGO-Y(A) en O(nlogn) qui retourne l'élément de A dont le nombre d'occurrences est minimal.

```
Algo-X(int[]A, int n){
int x = 0 ;
int y = 0 ;
for( int i = 1; i < n ; i++) {
   k = 1 ;
   for (int j = i + 1; j < n ; j++)
        if A[i] == A[j]
             k ++ ;
   if (x < k) {
        x = k;
        y = A[i];
 }
return y ;
}
```

Exercice 20

On considère un ensemble S de n (n≥2) entiers distincts stockés dans un tableau (S n'est pas supposé trié).

1. Proposer un algorithme en O(n) pour trouver deux éléments x et y de S tels que |x − y| soit maximal.

2. Proposer un algorithme en O(n log n) pour trouver deux éléments x et y de S tels que $x \neq y$ et $|x - y|$ soit minimal.
3. On dit qu'un élément x est majoritaire dans S si et si le nombre d'occurrences de x dans S est supérieur ou égal à n/2. Donner un algorithme permettant de chercher s'il existe un élément majoritaire dans S.

Exercice 21

On dit qu'un nœud dans un arbre binaire de recherche est plein s'il a à la fois un fils gauche et un fils droit.

1) Ecrivez un algorithme appelé Count-Full-Nodes (t) qui prend en entrée un arbre binaire de recherche enraciné
au nœud t, et retourne le nombre de nœuds pleins dans l'arbre.
Analysez la complexité de votre solution.

2) Ecrivez un algorithme appelé No-Full-Nodes (t) qui prend en entrée un arbre binaire de recherche enraciné à
t, et modifie l'arborescence sur place, en utilisant uniquement des rotations, de sorte que l'arborescence ne contienne aucun noud complet. Analysez la complexité de votre solution.

Exercice 22

On considère un ensemble S de n ($n \geq 2$) entiers distincts stockés dans un tableau (S n'est pas supposé trié).

4. Proposer un algorithme en O(n) pour trouver deux éléments x et y de S tels que $|x - y|$ soit maximal.
5. Proposer un algorithme en O(n log n) pour trouver deux éléments x et y de S tels que $x \neq y$ et $|x - y|$ soit minimal.
6. On dit qu'un élément x est majoritaire dans S si et si le nombre d'occurrences de x dans S est supérieur ou égal à n/2. Donner un algorithme permettant de chercher s'il existe un élément majoritaire dans S.

Exercice 23

L'algorithme ci-contre permet de calculer le graphe symétrique d'un graphe G=(E, Γ) où E est l'ensemble de sommets et Γ est la fonction successeur. Soit n le nombre de sommets du graphe G et m le nombre de ses arcs. Le résultat retourné par cet algorithme est alors la fonction prédécesseur Γ^{-1}. Supposons que tous les ensembles sont implémentés par des tableaux booléens.

```
Algorithme Sym (graphe G)
pour chaque x ∈ E faire Γ-1 (x) ← ∅ ;
pour chaque y ∈ E faire
        pour chaque x ∈ Γ(y) faire
        Γ-1 (x) ← Γ-1 (x) ∪ {y} ;
Fin.
```

1) Evaluer la complexité de cet algorithme en fonction de n et m.
2) Déduire un algorithme permettant de tester si le graphe G est symétrique puis évaluer sa complexité.
3) Ecrire un algorithme permettant de calculer un chemin entre deux sommets x et y de G puis évaluer sa complexité.

Exercice 24

1) Exprimer la complexité temporelle de chacun des 2 algos suivants en notation Θ :

ALGO1

```
int f (int A[] , x , y) {
```

```
        If (y = x+1)
                Return  A[x] ≤ A[y]
        Else
                { i = int ((x+y)/2) ;
                  Return f ( A ,x , i) && f ( A , i , y) ; }}
f(A, 0 , n) ;
```

```
ALGO 2
Sum = 0 ;
for ( i=1; i < n; i += 2 ) {
        for ( j = n; j > 0; j /= 2 ) {
                for ( k = j; k < n; k *= 2 ) {
                        sum += (i + j * k ); } } }
```

2) Un algorithme quadratique met 1 ms pour traiter 100 éléments de données. Combien de temps mettra-t-il pour le traitement de 5000 éléments de données.

Exercice 25

Pour chaque paire d'algos A et B, indiquer lequel est plus efficace en justifiant votre réponse :

1) Algorithme A en $O(n^2)$ et Algorithme B en $\Theta(n^2)$;
2) Algorithme A en $O(n^2)$ et Algorithme B en $\Theta(n\log n)$;
3) Algorithme B en $\Omega(n)$ et Algorithme B en $\Theta(n\log n)$;

Exercice 26

Considérons un algorithme Find-Elements-At-Distance(A, k) qui prend un tableau A de n entiers triés dans un ordre croissant et renvoie true si et seulement si A contient deux éléments A[i] et A[j] tels que $A[i] - A[j] = k$.

1) Écrire une version de l'algorithme Find-Elements-At-Distance de temps O(nlogn). Analysez brièvement la complexité de votre solution.
2) Écrivez une version de l'algorithme Find-Elements-At-Distance de temps O(n). Analysez brièvement la complexité de votre solution.

Exercice 27

Ecrire un algorithme Is-Perfectly-Balanced qui teste si un arbre binaire est parfaitement équilibré puis évaluer sa complexité.
Écrire un algorithme appelé Count-Full-Nodes(t) qui renvoie le nombre de nœuds complets dans l'arbre. Analysez la complexité de votre solution.

Exercice 28

Étant donné deux tableaux A et B, contenant tous deux n nombres, existent-t-ils un nombre k et une permutation A′ de A tels que $A'[i] + B[i] = k$ pour toutes les positions $i \in \{1, \ldots, n\}$.
1) Le problème est-il dans NP ? Écrivez un algorithme qui le prouve, ou argumentez le contraire.
2) Le problème est-il en P ? Écrivez un algorithme qui le prouve, ou argumentez le contraire.

Exercice 29

Écrivez un algorithme appelé Minimal-Contiguous-Sum(A) qui prend un tableau A de nombres, et retourne la valeur de la somme minimale des sous-séquences contiguës dans le temps O(n). Une somme de sous-équences contiguës est la somme de certains éléments contigus de A. Par exemple, si A est la séquence −1, 2,−2,−4, 1,−2, 5 − 2 − 3, 1, 2,−1 alors la sous-séquence contiguë minimale sum est −7, qui est la somme des éléments −2,−4, 1,−2.

CHAPITRE III. ELEMENTS D'OPTIMISATION COMBINATOIRE

Plan

1. Introduction
2. Définitions
3. Types de problèmes
4. Processus de résolution
5. Méthodes de résolution
6. Exemples de problèmes modèles

1. Introduction

Optimiser = maximiser un profit ou minimiser un coût = trouver le maximum ou le minimum
Optimiser un système = trouver sa meilleure configuration tout en exploitant les ressources disponibles (en respectant les contraintes).

Exemples

i. Problème de découpe maximale : minimiser les déchets ;
ii. Problème de transport : trouver le chemin minimal ;
iii. Bin packing : maximiser le profit ;
iv. Ordonnancement de tâches : minimiser le retard d'exécution ;
v. Gestion de projet : minimiser la durée totale d'exécution ;
vi. Emploi du temps : minimiser le nombre de salles, d'enseignants, la date de fin ;…

2. Définitions

RO recherche opérationnelle

Discipline des méthodes scientifiques pour aider à mieux décider.

- Objectif de la « RO » : faire de la recherche scientifique « opérationnelle » – utilisable sur « le terrain des opérations » – à l'aide des outils de l'informatique.
- Mettre au point des méthodes, les implémenter à l'aide d'outils (logiciels) pour trouver des résultats ensuite confrontés à la réalité (et repris jusqu'à satisfaction du demandeur).

Un problème d'Optimisation (PO) consiste à rechercher la valeur d'une variable x dans un ensemble S pour la laquelle la fonction *f*: S→R soit optimale (minimale ou maximale) tout en respectant certaines clauses C(x) et s'écrit (formulation mathématique) :

$$\begin{cases} Opt_{x \in S} f(x) \\ Sujet\ à\ C(x) \end{cases}$$

✓ x est la variable (de décision) du PO (il peut être entière, float, un vecteur, une matrice , une chaine, une image, une vidéo, un programme, un emploi de temps, une stratégie d'achat, ...).
✓ Opt. = optimiser = minimiser ou maximiser = opérateur = trouver le max ou le min.
✓ *f* s'appelle l'objectif ou le critère ou encore la fonction économique à optimiser ;
✓ S est dit espace de recherche ou espace d'états ou de configurations.

✓ C(x) représente une ou plusieurs assertions dites contraintes du PO.
✓ Tout x de S est dit solution candidate du PO.
✓ Si x satisfait les contraintes C(x), x est dit une solution admissible, réalisable ou faisable du PO (x∈A, figure III.1.).
✓ Si $f(x_0) \approx f(x^*)$ c.-à-d. $|f(x_0)-f(x^*)| < \varepsilon$ (ε donné) alors x_0 est dite solution approchée du PO. (le contraire: solution exacte c.-à-d. optimale)

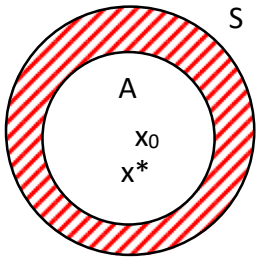


Figure III.1. Espace de recherche et solutions admissibles.

✓ Le voisinage d'une solution x_0 noté V_{x_0} est l'ensemble de solutions x telles que $f(x) \approx f(x_0)$ c.à-d. $|f(x_0) - f(x)| < \varepsilon$.
✓ Le PO possède un minimum local en x_0 si et seulement si $\forall x \in V_{x_0}: f(x) \leq f(x_0)$.
✓ Le PO possède un minimum global en x* si et seulement si $\forall x \in S: f(x) \leq f(x_0)$. (On peut définir les maximums local et global de façon analogique).

Exemple 1

$$\begin{cases} Min_{x\in[1,25]} f(x) \\ Sujet\ à\ y \geq 1.5\ ;\ x \in R \end{cases}$$

où f est représentée par la figure III.2 ci-dessous.

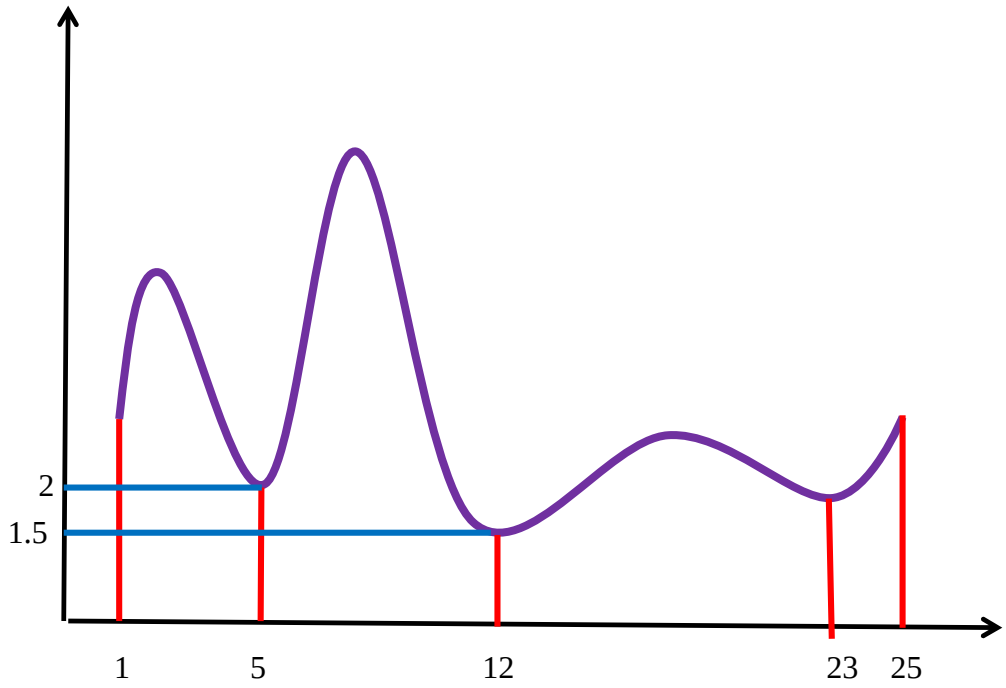


Figure III.2. Représentation graphique de la fonction f de l'exemple1.

Espace de recherche S = [1,25]
Solution optimale x* =12
$f(12) = 1.5\ estun$ minimum global.
$f(5) et\ f(23)\ sont\ des$ minima locaux.
$[4.99, 5.10]$ est un voisinage de 5.
$5,23;\ 11.9;\ 12.1$ sont des solutions approchées.
1 est une solution candidate mais non faisable.

Exemple 2

Une entreprise fabrique deux types de produits A et B.
La réalisation d'une unité de A nécessite 30 DA de matière première et 120 DA de main d'œuvre.
La réalisation d'une unité de B nécessite 60 DA de matière première et 80 DA de main d'œuvre.
Les profits réalisés sont de 50 DA par unité de A, et de 40 DA par unité de B.
Les contraintes économiques font que :

- La dépense journalière en matière première ne doit pas dépasser 580 DA.
- La dépense journalière en main d'œuvre ne doit pas dépasser 1250 DA.

On désire trouver le plan de production journalière (c.-à-d. les nombres d'unités de A et B à fabriquer par jour) qui maximise le profit journalier de l'entreprise.

1. Donner une formulation mathématique à ce problème.
2. Quel est le type de ce PO ?
3. Expliciter l'espace de recherche ; la fonction objectif ; les contraintes ;

Solution

Soit x1 le nombre d'unités du produit A à fabriquer;
Soit x2 le nombre d'unités du produit B à fabriquer;
La formulation mathématique de ce problème d'optimisation s'écrit comme suit:

$$\begin{cases} Max\ f(x_1, x_2) = 50x_1 + 40x_2 \\ 30x_1 + 60x_2 \leq 580 \\ 120x_1 + 80x_2 \leq 1250 \\ x_1 \in N, x_2 \in N \end{cases}$$

La variable x = $(x_1, x_2) \in N^2$.
L'objectif $f(x_1, x_2) = 50x_1 + 40x_2$.
L'espace de recherche est N^2.
(on reviendra plus loin dans ce cours aux méthodes de résolution de ces problèmes).

3. Types de problèmes d'optimisation

Soit un problème d'optimisation :

$$\begin{cases} Opt_{x \in S} f(x) \\ Sujet\ à\ C(x) \end{cases}$$

- ✓ Si x est continu (S est continu), le PO est dit continu ;
- ✓ Si x est discret (S est discret), le PO est dit discret ;
- ✓ Si le PO est discret et S est fini ($|S| \in N$) alors le PO est Combinatoire ;
  (souvent, |S| est assez grand c.-à-d. |S| est en $O(2^n)$, c'est ce qu'on appelle explosion combinatoire).

- ✓ Si x est une fonction de temps, le PO est dit dynamique ; (le contraire statique).
- ✓ Si x dépend d'une probabilité (aléatoire), le PO est dit stochastique ; (le contraire déterministe)
- ✓ Si f est un vecteur, c.-à-d. f : S → R^n, le PO est dit multiobjectifs ou multicritères ; (le contraire monoobjectif ou monocritère).

Exemple

Dans les deux exemples ci-dessus, le premier est un problème continu tant disque le second est discret, et les deux cas sont monoobjectifs et déterministes.

4. Processus de resolution d'un PO

Pour aborder un PO, il ne suffit pas d'écrire directement un programme permettant de donner une solution pour toute instance de l'input à ce problème, mais il s'agit d'un processus rigoureux et minutieux à suivre en plusieurs étapes (figure III.3) :

i. L'énoncé du PO : il faut bien spécifier le problème en citant clairement son input, son output, l'objectif à optimiser, l'espace de recherche et éventuellement ses contraintes.
ii. La formulation mathématique : exprimer les éléments ci-dessus à l'aide d'outils mathématiques (vecteurs, matrices, équations, inéquations, assertions, …).
iii. Modélisation : Exprimer le PO en fonction de l'un des méta problèmes connus afin de faciliter son traitement.
iv. Classification : Etudier la complexité du PO afin de justifier la méthode à utiliser (exacte ou approchée par exemple).
v. Implémentation : choisir l'environnement (langage, OS, hardware, …) puis implémenter la méthode adoptée.

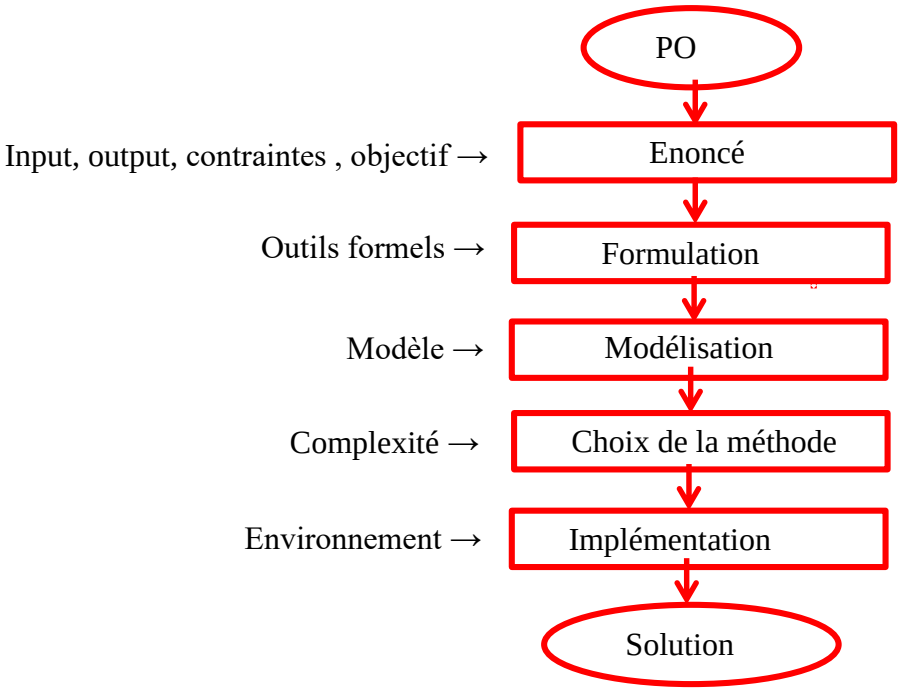


Figure III.3. Processus de résolution d'un PO.

5. Méthodes de résolution d'un PO

En général, la résolution d'un problème consiste à explorer l'espace de recherche pour dégager la solution optimale (c'est comme par exemple la recherche d'un élément max ou min dans une liste).

Une méthode de résolution est donc une stratégie d'exploration de l'espace de recherche (c'est ce qui caractérise chaque méthode).

D'où la classification récapitulée dans le tableau ci-dessous :

Méthodes de résolution	
Exactes	**Approchées**
Programmation dynamique Méthode par séparation et évaluation	Heuristiques Métaheuristiques
Algorithmes gloutons	

Méthode exacte = méthode qui fournit une solution exacte, c'est une exploration totale de l'espace de recherche.

Méthode approchée = méthode qui fournit une solution approchée c'est une exploration partielle (donc stochastique mais guidée) de l'espace de recherche.

Une heuristique : est une méthode approchée spécifique applicable à un PO déterminé.

Une métaheuristique : est une méthode approchée générique applicable à tout PO.

Algorithme glouton : un algorithme qui construit une solution par étapes, à chaque étape, la décision optimale est prise sans tenir compte du sort de la solution globale.

Chronologie d'évolution des méthodes:

- → Fin des années 1940 : concurrence vers l'exact ; Simplexe, G. Dantzig (1947) ; PD, R. Bellman (1956).
- → Début des années 1980 : avènement des métaheuristiques ; RT, Glover (1989) ; AG, J. Holland (1975). D. Goldberg en 1989. PSO, Russel Eberhart (1995); RT, Glover (1986), …
- → Années 1990 : l'ère de l'hybridation et méthodes émergentes; Martin et Otto, descente dans le RS (1990) ; Stûtzle et Hoos , RL dans les CF , (2000) ; Talbi , RT dans un AG (1998).

6. Quelques exemples de problèmes

Les 21 problèmes NP-complets de Karp (RICHARD KARP 1972)

Les 21 problèmes NP-complets de Karp ont marqué une étape importante de l'histoire de la théorie de la complexité des algorithmes. Ce sont 21 problèmes réputés difficiles de combinatoire et de théorie des graphes qui sont réductibles entre eux. C'est ce qu'a démontré Richard Karp en 1972 dans son article Reducibility Among Combinatorial Problems, de même que leur NP-complétude. Un des plus importants résultats en théorie de la complexité est celui de Stephen Cook, en 1971. Dans son article, il montre le premier problème NP-complet, soit le problème SAT (voir théorème de Cook). C'est cette idée que Karp développe en l'appliquant à des problèmes de combinatoire et de théorie des graphes.

Les 21 problèmes sont organisés en indentations de façon à indiquer la direction de la réduction servant à prouver leur NP-complétude. Par exemple, le problème du sac à dos a été prouvé NP-complet par une réduction à partir de celui de la couverture exacte. Le nom anglais original est en lettres capitales.

- SATISFIABILITY : le problème SAT pour les formules en forme normale conjonctive
    - CLIQUE : le problème de la clique (voir aussi le problème de l'ensemble indépendant)
        - SET PACKING : Set packing (empaquetage d'ensemble)
        - VERTEX COVER : le problème de couverture par sommets
            - SET COVERING : le problème de couverture par ensembles
            - FEEDBACK ARC SET : feedback arc set
            - FEEDBACK NODE SET : feedback vertex set
            - DIRECTED HAMILTONIAN CIRCUIT : voir graphe hamiltonien
                - UNDIRECTED HAMILTONIAN CIRCUIT : voir graphe hamiltonien
    - 0-1 INTEGER PROGRAMMING : voir optimisation linéaire en nombres entiers
    - 3-SAT : voir problème 3-SAT
        - CHROMATIC NUMBER : coloration de graphe
            - CLIQUE COVER : partition en cliques
            - EXACT COVER : couverture exacte
                - MATCHING à 3 dimensions : appariement à 3 dimensions
                - STEINER TREE : voir arbre de Steiner
                - HITTING SET : ensemble intersectant
                - KNAPSACK : problème du sac à dos
                    - JOB SEQUENCING : séquençage de tâches
                    - PARTITION : problème de partition
                        - MAX-CUT : problème de la coupe maximum

Remarque

Ces problèmes sont souvent utilisés comme modèles pour classer et traiter d'autres problèmes et sont en effet dits méta problèmes.

Théorème

Pour tout problème d'optimisation, il existe un problème de décision lui est équivalent.

Résultat

La classification des problèmes de décision demeure valable pour les problèmes d'optimisation.

Exemple

« *Trouver le plus court chemin* » est un problème d'optimisation, le problème de décision qui lui est équivalent peut s'énoncer : « *Existe-t-il un chemin de longueur l* ?».

Le problème SAT

"satisfiabilité" d'une expression logique.

Exemple

$f(x,y,z,t) = (x \vee y \vee z) \wedge (x \vee y \vee \bar{t}) \wedge (y \vee t) \wedge (\bar{x} \vee z)$

Peut-on affecter des valeurs vrai ou faux aux variables de telle façon que l'expression soit vraie ?

Exemple

Une solution: x=vrai y=faux t=vrai z=vrai

En y a 2^n solutions candidates, n est le nombre de variables (n= 4 et 2^n=16 pour l'exemple ci-dessus).

SAT= SATISFIABILITE (SATISFIABILITY en anglais).

CNF = Conjonctive Normal Form, l'expression booléenne est une conjonction de clauses ;

DNF = Disjonctive Normal Form, l'expression booléenne est une disjonction de clauses ;

Une clause = une disjonction de littéraux ;

Un littéral = x ou $\bar{x}$;

SAT-K : chaque clause contient au plus k littéraux ;

Exemples

Notation	Exemple
SAT-3 CNF	$f(x,y,z,t) = (x \vee y \vee z) \wedge (x \vee y \vee \bar{t}) \wedge (y \vee t) \wedge (\bar{x} \vee z)$
SAT-3 DNF	$f(x,y,z,t) = (x \vee y \vee z) \vee (x \vee y \vee \bar{t}) \vee (x \vee z \vee t) \vee (\bar{x} \vee z \vee \bar{t})$
SAT-2 CNF	$f(x,y,z,t) = (x \vee y) \wedge (x \vee \bar{t}) \wedge (t) \wedge (\bar{x} \vee z)$
SAT	$f(x,y,z,t) = x \vee y \vee z \wedge x \vee \bar{t} \wedge \bar{x} \vee z$

Théorème

SAT-DNF $\in$ P

SAT-2, sat-1 $\in$ P

SAT–k (CNF)$\in$ NP-C pour k > 2

Toute instance de SAT–k (CNF) peut être réduit polynomialement à SAT-3 (CNF).

Remarque

En facto, le cas le plus fréquent est SAT-CNF qui est le premier problème prouve NP-complet. (Exemple d'application : test des portes logiques d'un processeur).

Le théorème de Cook

Stephen Cook a classé le problème SAT comme *NP*-complet

SAT est le premier problème *NP*-complet connu

PROBLÈME DU SAC A DOS (KNAPSACK PROBLEM KSP)

Étant donné plusieurs objets Oi ; i=1,n ; possédant chacun un poids wi et une valeur vi et étant donné un poids maximum pour le sac W, quels objets faut-il mettre dans le sac de manière à maximiser la valeur totale sans dépasser le poids maximal W autorisé pour le sac ?

En y a 2^n solutions candidates

KSP est NP-difficile au sens faible (Weakly hard), Pseudo polynomial.

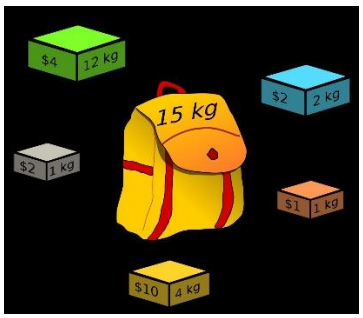


Figure III.4. Exemple de KSP.

Formulation mathématique

Une solution x peut être représentée par un vecteur binaire (x_i) i=1,n tel que $x_i = 1$ *si l'objet i est pris dans le sac* et $x_i = 0$ *sinon.*

$$\begin{cases} max \sum_{i=1}^{n} x_i v_i \\ \sum_{i=1}^{n} x_i w_i \leq W \\ x_i \in \{0,1\}, i = 1, n \end{cases}$$

Exemple

$n = 4$ et un sac à dos d'un poids maximal de 30 kg ($W = 30$), nous avons par exemple les données suivantes :

Objets	1	2	3	4
v_i	7	4	3	3
w_i	13	12	8	10

Ensuite, il nous faut définir les variables qui représentent en quelque sorte les actions ou les décisions qui amèneront à trouver une solution. On définit la variable x_i associée à un objet *i* de la façon suivante : $x_i = 1$ si l'objet *i* est mis dans le sac, et $x_i = 0$ si l'objet *i* n'est pas mis dans le sac.

Dans notre exemple, une solution réalisable est de mettre tous les objets dans le sac à dos sauf le premier, nous avons donc : $x_1 = 0$, $x_2 = 1$, $x_3 = 1$, et $x_4 = 1$.

Variantes du KSP

Il est à noter qu'il existe plusieurs variantes[1] du problème du sac a dos qui se diffèrent par la nature de la variable, la fonction objectif et/ou la ou les contraintes liées au sac.

Simple 01KSP

La variante détaillée ci-dessus est la plus simple et dite simple ou encore 0/1 et notée 01KSP.

Quadratique QKSP

Une valeur supplémentaire s_{ij} lorsque les objets i et j sont prix simultanément. Ainsi la fonction objectif devient alors : $max\ (\sum_{i=1}^{n} x_i v_i + \sum_{i=1}^{n} x_i x_j s_{ij}$.

Multiple MSKP

On dispose de plusieurs, m par exemple, sacs de de capacités respectives $W_k, k = 1, m$. Le problème aura alors k contraintes de la forme : $\sum_{i=1}^{n} x_i w_i \leq W_k, k = 1, m$.

[1] https://interstices.info/le-probleme-du-sac-a-dos/

Multidimensionnel d-KSP

Le sac possède plusieurs, d par exemple, dimensions $W_k, k = 1, d$ et chaque objet i possède également d poids $w_{ij}, j = 1, d$ (un poids sur chaque dimension). Le problème aura alors d contraintes de la forme : $\sum_{i=1}^{n} x_i w_{ij} \leq W_j, j = 1, d.$.

A variables continues ou Fractionnaire

On peut prendre une fraction d'un objet.

A variables entières

On peut prendre plusieurs exemplaires d'un même objet.

Multiobjectif

Chaque objet dispose de plusieurs valeurs.

Quiz : Ecrire la formulation mathématique de chacune des 3 dernières variantes.

KSP / APPLICATIONS

i. Trouver le moyen le moins coûteux de réduire les matières premières
ii. Sélection d'investissements et de portefeuilles
iii. Sélection d'actifs pour la titrisation adossée à des actifs
iv. Générer des clés pour le Merkle – Hellman et d'autres crypto systèmes à dos.

PROBLEME DU VOYAGEUR DE COMMERCE (PVC)

TRAVELLING SALLESMAN PROBLEM (TSP)

Enoncé

Un voyageur de commerce doit visiter un nombre fini de villes et revenir à son point d'origine en passant par chaque ville une et une seule fois. Trouvez l'ordre de visite des villes qui minimise la distance totale parcourue par le voyageur[2].

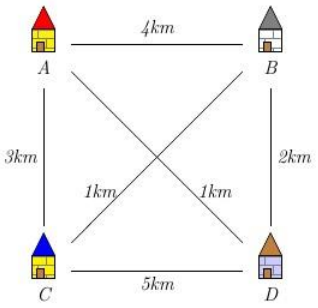


Figure III.5. Exemple de TSP.

Formulation mathématique

[2] www.tsp.com www.satlib.com

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}:$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1 \qquad j = 1, \ldots, n;$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1 \qquad i = 1, \ldots, n;$$

$$\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1 \qquad \forall Q \subsetneq \{1, \ldots, n\}, |Q| \geq 2$$

Le PVC peut être modélisé à l'aide d'un graphe constitué d'un ensemble de sommets et d'un ensemble d'arêtes. Chaque sommet représente une ville, une arête symbolise le passage d'une ville à une autre, et on lui associe un poids pouvant représenter une distance, un temps de parcours ou encore un coût.

Résoudre le problème du voyageur de commerce revient à trouver dans ce graphe un cycle passant par tous les sommets une unique fois (un tel cycle est dit « hamiltonien ») et qui soit de longueur minimale.

Pour un PVC symétrique de n villes : (n-1)!/2 solutions candidates

Pour un PVC asymétrique de n villes : (n-1)! solutions candidates Pour le graphe ci-contre, une solution à ce problème serait le cycle 1, 2, 3, 4 et 1, correspondant à une distance totale de 23. Cette solution est optimale, il n'en existe pas de meilleure.

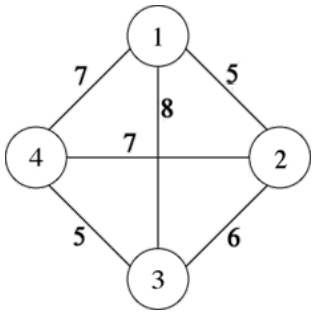


Figure III.6. Exemple de TSP.

Remarque

Les distances entre les villes du TSP peuvent être fournies sous l'une des formes :

Métrique : on donne la matrice des distances entre villes (matrice d'adjacence du graphe), on parle alors du PVC *métrique*.

Euclidienne : on donne les coordonnées des villes, on parle alors du PVC e*uclidien* (ce cas concerne uniquement le PVC *symétrique*).

PVC / applications

i. Perforation de circuits imprimés.
ii. Révision des moteurs à turbine à gaz.
iii. Cristallographie aux rayons X.
iv. Câblage informatique.
v. Le problème de la préparation des commandes dans les entrepôts.
vi. Itinéraire de véhicules.
vii. Traçage de masques dans la production de PCB.
viii. Planification et ordonnancement.

PROBLEME DE PARTITION PP (PARTITION PROBLEM PP)

Enoncé

Étant donné un multiensemble S d'entiers naturels, déterminer s'il existe une partition de S en deux sous-ensembles S_1 and S_2 tels que la somme des éléments de S_1 soit égale à la somme des éléments de S_2.

PROBLEME DE k-COLORATION DE GRAPHE (GRAPH COLORING)

Enoncé

La **coloration de graphe** consiste à attribuer une couleur à chacun de ses sommets de manière que deux sommets reliés par une arête soient de couleur différente. On cherche souvent à utiliser le nombre minimal k de couleurs, appelé **nombre chromatique (figure 9)**.

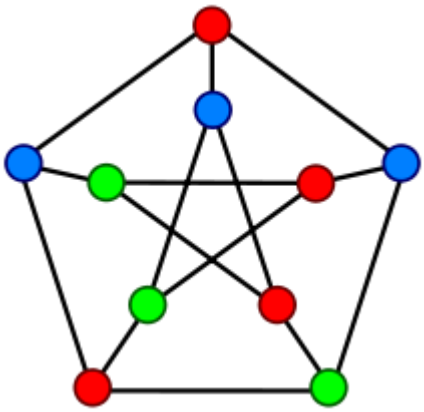


Figure III.7. Représentation d'une solution du problème 3-coloration.

PROBLEME DE L'ARBRE COUVRANT MINIMAL (MINIMUM SPANNING TREE)

Enoncé

Étant donné un graphe non orienté connexe dont les arêtes sont pondérées, un arbre couvrant de poids minimal (ACM) de ce graphe est un arbre couvrant (sous-ensemble qui est un arbre et qui connecte tous les sommets ensemble) dont la somme des poids des arêtes est minimale (c'est-à-dire de poids inférieur ou égal à celui de tous les autres arbres couvrants du graphe). L'arbre couvrant de poids minimal est aussi connu sous certains autres noms, tel qu'arbre couvrant minimum ou encore arbre sous-tendant minimum.

Exemple

Dans le graphe suivant (figure 10), l'arbre couvrant minimum est le sous graphe représenté en gras.

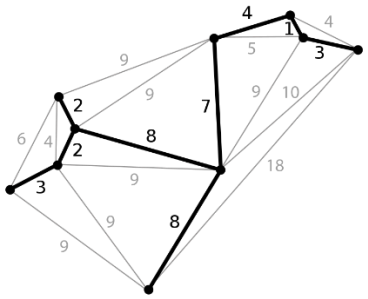
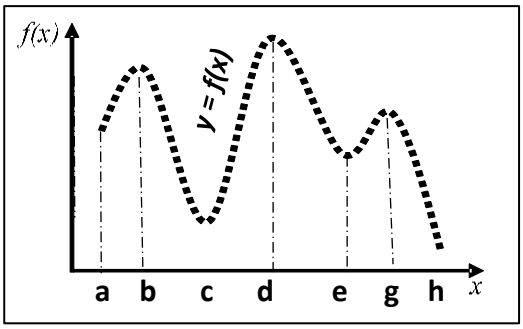


Figure III.8. Représentation de l'arbre couvrant minimal dans un graphe.

PROBLEME DE RENDU DE MONNAIE (CHANGE MAKING PROBLEM)

Enoncé

Étant donné un système de monnaie (pièces et billets), comment rendre une somme donnée M de façon optimale, c'est-à-dire avec le nombre minimal de pièces et billets ?



Formulation mathématique

Soit x_i le nombre d'exemplaires de la pièce i de valeur v_i :

$$\begin{cases} \min \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i v_i = M \\ x_i \in N \end{cases}$$

Quiz : calculer le nombre de solutions candidates.

Exemple

On dispose d'un jeu de pièces, par exemple [1, 2, 5, 10] et d'une somme à construire, par exemple S=12.

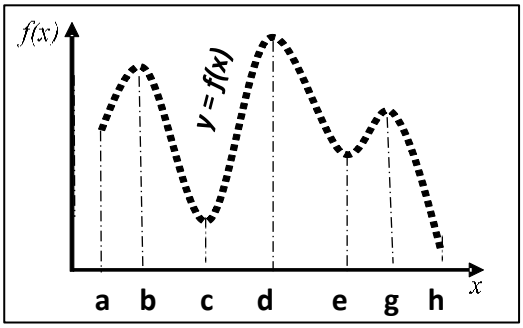
Quel est le nombre minimal de pièces nécessaires pour atteindre S ?

Dans notre exemple, la solution 12=10+2 convient. C'est aussi la solution optimale, il faut donc deux pièces : [10, 2].

7. Exercices

Exercice 1

La figure ci-dessous illustre la courbe d'une fonction entière *f* d'une variable entière *x* dont on se propose de maximiser.



1) Ecrire la formulation mathématique de ce pb.
2) Quel est le type de ce pb.
3) Calculer l'espace de recherche et sa taille.
4) Déterminer sa solution optimale.
5) Que représentent b et g pour ce pb.

Exercice 2

Considérons le PVC (TSP) symétrique et métrique avec n le nombre de villes et d[][] la matrice des distances.

1) Comment représenter une solution pour ce problème.
2) Ecrire la formulation mathématique de ce pb.
3) Rappeler le nombre de solutions candidates.
4) Ecrire l'algorithme retournant la distance totale d'une solution x puis évaluer sa complexité.
5) Ecrire un algorithme exhaustif pour résoudre ce pb.

Exercice 3

Considérons le PSD (KSP) simple (0/1) (n, vi, wi, W).

1) Comment représenter une solution pour ce problème.
2) Ecrire la formulation mathématique de ce pb.
3) Rappeler le nombre de solutions candidates.
4) Ecrire l'algorithme qui teste si un algorithme est admissible puis évaluer sa complexité.
5) Ecrire l'algorithme retournant la valeur totale d'une solution x puis évaluer sa complexité.
6) Ecrire un algorithme exhaustif pour résoudre ce pb.

Exercice 4

Considérons le problème SAT-k d'une fonction booléenne f de m clauses et n variables définie par une matrice m×k.

1) Rappeler le nombre de solutions candidates.
2) Ecrire l'algorithme qui teste si une solution candidate satisfait la fonction booléenne.
3) Ecrire un algorithme polynomial qui résout CNF-2.
4) Ecrire un algorithme polynomial qui résout DNF-k.
5) Ecrire un algorithme exhaustif qui résout CNF-k.

Exercice 5

Etant donnés n articles chacun ayant un revenu de transport v_i dinars et un poids w_i kg. On désire déterminer quels articles transporter par un camion de capacite C kg ($C < \sum_{i=0}^{n-1} w_i$) afin de maximiser le revenu total de transport.

1) Quel problème vu au cours pouvant être considéré comme modèle pour ce problème ?
2) Déduire la classe de complexité de ce problème.
3) Ecrire la formulation mathématique de ce problème.
4) Calculer la taille de l'espace de recherche.
5) L'algorithme ci-contre constitue une partie de la résolution de ce problème.
   a) Que représente le vecteur binaire x pour le problème donné.
   b) Que fait cet algorithme ? Evaluer sa complexité.
6) Ecrire l'algorithme qui calcule le revenu total de transport pour une solution x puis évaluer sa complexité.

```
boolean  ALGO (int n , int [] w , binary [] x , int C)
{  s = 0 ;
   for ( i = 0 ; i < n ; i++)
            s + = x[i] * w[i] ;
   return (s <= C) ;  }
```

Exercice 6

Une entreprise fabrique 2 types de produits P_1 et P_2 .
Pour produire une unité de P_1 , l'entreprise utilise 3 kg de matière première.
Pour produire une unité de P_2 , l'entreprise utilise 2 kg de matière première et une unité de P_1 .
On dispose de 500 kg de matière première par jour.
Le produit P_1 utilisé comme semi-fini peut également être vendu.
Les revenus unitaires de P_1 et P_2 sont de 40 et 60 DA respectivement.
L'objectif est de maximiser le revenu total des produits vendus par jour.

1) Formulez un modèle mathématique du problème.
2) Quel est le type de ce problème d'optimisation ?
3) Donner une solution admissible et calculer le revenu correspondant.

Exercice 7

Une entreprise fabrique 3 types de produits P_1 , P_2 et P_3 .
Pour produire 1 unité de produit P_1 , l'entreprise utilise 2 kg de matière première.
Pour produire 1 unité de produit P_2 , l'entreprise utilise 3 kg de matière première et 1 unité de produit P_1 .
Pour produire 1 unité de produit P_3 , l'entreprise utilise 4 kg de matière première, 3 unités de produit P_1 et 2 unités de produit P_2 . Il y a 1000 kg de matière première disponible.
Les produits P_1 et P_2 utilisés comme produits semi-finis peuvent également être vendus. Les prix des biens P_1 , P_2 et P_3 sont de 5, 10 et 30 DA. L'objectif est de maximiser le revenu total des produits vendus.

1) Formulez un modèle mathématique du problème.
2) Donner une solution admissible.
3) Quel est le type de ce problème d'optimisation ?

Exercice 8

Etant donné n entiers a_i ; i=1,n et un entier B. On se propose de trouver le sous-ensemble des entiers a_i dont la somme est maximale et ne dépassant pas B sachant que $B < \sum_{i=1}^{n} a_i$.
Exemple : input : n=4 ; a[]={2,4,3,5} ; B= 11 ; output : {2,4,5}

1) Ecrire la formulation mathématique de ce problème.
2) Donner l'espace de recherche de ce problème et sa taille.
3) Quel est le type de ce problème d'optimisation.
4) Montrer que ce problème est un cas particulier du problème du sac à dos.
5) En termes de complexité, que représente ce problème par rapport au problème du sac à dos ?
6) En déduire la classe de complexité de ce problème.
7) Appliquer la méthode de la programmation dynamique à l'exemple ci-dessus.
8) Ecrire l'algorithme correspondant et évaluer ses complexités temporelle et spatiale.

Exercice 9

Le problème du sac-à-dos à variables entières bornées s'énonce comme suit :
Etant donnés n objets chacun ayant une valeur v_i, un poids w_i et un nombre d'exemplaires n_i.
On dispose d'un sac de capacite C ($C < \sum_{i=0}^{n-1} n_i * w_i$).
On désire déterminer les nombres d'exemplaires de chaque objet à mettre dans le sac dont la somme pondérée des valeurs est maximale.

1) Ecrire la formulation mathématique de ce problème.
2) Calculer la taille de l'espace de recherche.
3) Proposer une heuristique pour calculer une borne supérieure et une autre pour calculer une borne inferieure de l'optimum (les algorithmes détaillés ne sont pas demandés, expliquer juste l'idée.)

Exercice 10

Soit le problème de coloration des arrêtes d'un graphe G(X, E) avec X l'ensemble des sommets et E l'ensemble des arrêtes. L'objectif est de colorer les arrêtes du graphe tel que deux arêtes ayant une extrémité commune soient de couleurs différentes. Soit ci la couleur de l'arrête i. Le but de l'exercice est d'appliquer la méthode de recherche taboue pour le problème.
Donnez la représentation d'une solution du problème, montrer à l'aide d'un graphe cette représentation.
Considérons trois couleurs c1, c2 et c3 et selon votre représentation, donnez une solution s initiale.

Exercice 11

Un ordinateur a une variété de composants à connecter par des fils. La distance en millimètres entre chaque paire de composants est donnée dans le tableau ci-dessous. On se propose de déterminer les paires de composants à connecter afin que l'ensemble des composants soit connecté et que la longueur totale du fil entre les composants soit minimisée.

	1	2	3	4	5	6
1	0	6,7	5.2	2,8	5,6	3,6
2	6,7	0	5,7	7,3	5.1	3.2
3	5.2	5,7	0	3.4	8.4	4.0
4	2,8	7,3	3.4	0	8,0	4.4
5	5,6	5.1	8.4	8,0	0	4.6
6	3,6	3.2	4.0	4.4	4.6	0

1. Ecrire la formulation mathématique de ce problème.
2. Donne la taille de l'espace de recherche.
3. Quel est le type de ce problème.
4. Quel est le modèle vu au cours pour ce problème.
5. Proposer une heuristique de résolution de ce problème.
6. Ecrire l'algorithme correspondant et évaluer sa complexité.
7. Appliquer cet algorithme à l'instance donnée.

Exercice 12

Une entreprise fabrique 3 types de produits P_1, P_2 et P_3.
Pour produire 1 unité de produit P_1, l'entreprise utilise 3 kg de matière.
Pour produire 1 unité de produit P_2, l'entreprise utilise 2 kg de matière et 1 unité de produit P_1.
Pour produire 1 unité de produit P_3, l'entreprise utilise 2 kg de matière, 2 unités de produit P_1 et 1 unité de produit P_2. Il y a 1000 kg de matériel disponible.
Les produits P_1 et P_2 utilisés comme produits semi-finis peuvent également être vendus eux-mêmes. Les prix des biens P_1, P_2 et P_3 sont de 5, 10 et 30 e.
L'objectif est de maximiser les revenus totaux des produits vendus.

1. Ecrire la formulation mathématique de ce problème.
2. Donne la taille de l'espace de recherche.
3. Quel est le type de ce problème.
4. Quel est le modèle vu au cours pour ce problème.
5. Proposer une heuristique de résolution de ce problème.
6. Ecrire l'algorithme correspondant et évaluer sa complexité.
7. Appliquer cet algorithme à l'instance donnée.

Exercice 13

Problème d'affectation linéaire

Une course de relais pour des équipes de 5 membres est organisée. Un membre de chaque équipe concourra dans une discipline. Vous allez constituer une équipe la plus solide. Dans le tableau, les meilleures performances saisonnières (en minutes) des candidats sont données.

SB	Footing	Natation	Cyclisme	Saut	Ski
1	75	25	202	130	165
2	87	24	198	127	173
3	68	19	195	121	164
4	91	20	207	122	182
5	80	28	215	125	172
6	78	22	197	125	180
7	75	25	205	127	178
8	81	23	211	131	165

Formuler ce problème.

Exercice 14

Problème du sac à dos

Il y a 5 projets caractérisés par leurs coûts d'investissement et leurs revenus. Le budget de 50 000 est disponible pour sélectionner les projets qui assurent le rendement total le plus élevé.

	P1	P2	P3	P4	P5
Coût	12 000	10 000	15 000	18 000	16 000
Revenu	20 000	18 000	22 000	26 000	21 000

Proposer une heuristique de résolution puis donner la solution pour cette instance.
Donner la solution dans le cas fractionnaire.

Exercice 15

Une entreprise produit des clôtures en lattes de jardin. Il n'y a que des lattes standard de 200 cm de long à disposition dans l'entrepôt. Pour produire une clôture, l'entreprise a besoin d'exactement 1200 lattes de 80 cm de long, 3100 lattes de 50 cm de long et 2100 lattes de 30 cm de long.
Vous devez concevoir un plan de coupe pour minimiser la quantité totale de lattes de 200 cm de long.
Formulez un modèle mathématique du problème.
Quel est le type de ce problème ?

A quelle classe de complexité appartient-il ?

Exercice 16

Un projet se compose de 5 parties indépendantes. Dans l'entreprise, 5 départements peuvent gérer les parties individuellement. Les données historiques montrent le temps moyen (en jours) que les départements ont terminé des tâches similaires (voir le tableau). NA représente le fait qu'un département n'a pas travaillé sur une telle tâche dans le passé. L'entreprise souhaite terminer l'ensemble du projet le plus rapidement possible.

Temps	Partie 1	Partie 2	Partie3	Partie 4	Partie5
Dépt1	25	15	N / A	17	25
Dépt2	22	N / A	22	20	22
Dépt3	20	18	25	16	23
Dépt4	N / A	20	30	21	28
Dépt5	27	19	27	18	N / A

Ecrire la formulation mathématique.

Modéliser ce problème à l'aide d'un graphe et construire ce graphe.

Exercice 17

Problème d'emplacement d'installations

L'entreprise utilise 7 entrepôts potentiels pour ses 5 filiales. Dans le tableau ci-dessous, les besoins mensuels des filiales et les capacités mensuelles des entrepôts sont indiqués (en milliers de tonnes). Si un entrepôt est utilisé, l'entreprise doit payer un loyer mensuel (en milliers). De plus, le coût unitaire de transport (par tonne) est calculé pour chaque paire d'entrepôt et filiale. Quel entrepôt utiliser et quelles quantités de matériel transporter entre les entrepôts et les filiales. L'objectif est de minimiser le coût mensuel total.

FLP	SD1	SD2	SD3	SD4	SD5	Cap	Rent
WH1	10	15	20	12	8	20	10
WH2	7	10	15	22	13	25	12
WH3	20	13	10	11	9	15	8
WH4	15	12	21	18	16	18	9
WH5	11	22	12	10	15	22	11
WH6	9	13	11	18	22	30	13
WH7	18	10	15	7	9	23	11
Req	25	22	17	22	15		

Donner la formulation mathématique du problème.

Exercice 18

Problème de rangement

Les produits doivent être transportés chez le client dans des conteneurs identiques. Dans le tableau, un poids unitaire de chaque type de produit (en kg) et un nombre d'entre eux à transporter sont indiqués. La capacité de poids du conteneur est de 500 kg. L'objectif est de minimiser le nombre de conteneurs usagés.

Bin Packing	Poids	Nombre
Produit1	20	13
Produit2	22	15
Produit3	18	25
Produit4	15	30
Produit5	21	18
Produit6	16	35

Donner la formulation mathématique du problème.

Exercice 19

Problème de débit maximum

Trouver le débit maximum du nœud 1 au nœud 6 pour le graphe donné par le tableau suivant

Arc	Capacité	Arc	Capacité
(1,2)	10	(3,5)	7
(1,3)	10	(3,6)	5
(1,4)	12	(4,3)	3
(2,5)	11	(4,6)	9
(3,4)	3	(5,6)	18

Arc	Capacité	Coût	Arc	Capacité	Coût
(1,2)	10	5	(3,5)	7	6
(1,3)	10	10	(3,6)	5	9
(1,4)	12	20	(4,3)	3	12
(2,5)	11	11	(4,6)	9	17
(3,4)	3	12	(5,6)	18	8

Ecrire la formulation mathématique.

Exercice 20

Arbre couvrant minimal

L'entreprise doit installer 6 panneaux d'information dans le parc de la ville. Ils doivent être reliés par un câble passant sous les trottoirs. Les distances (en 10 mètres) entre les planches sont indiquées dans le tableau. S'il n'y a pas de chaussée entre une paire de planches, une valeur prohibitive 100 est définie. L'objectif est de minimiser le coût total des travaux d'excavation et du câble lui-même.

Planches	1	2	3	4	5	6
1	0	6	5	100	100	100
2	6	0	7	2	4	100
3	5	7	0	6	100	8
4	100	2	6	0	3	4
5	100	4	100	3	0	5
6	100	100	8	4	5	0

Ecrire la formulation mathématique.

Exercice 21

Arbre de Steiner minimal

Trois utilisateurs (nœuds 2, 3 et 4) doivent être connectés à l'émetteur (nœud 1) soit directement, soit via deux stations de transfert (nœuds 5 et 6). Dans le tableau, les valeurs de coût (en milliers d'e par mois) pour les connexions possibles sont données. L'utilisation des stations de transfert est facturée 30 et 20 milliers par mois. Trouver la valeur optimale de connexion.

Arc	Coût	Arc	Coût
(2,1)	15	(4,5)	9
(2,5)	3	(4,6)	6
(3,1)	18	(5,1)	7
(3,5)	4	(6,1)	12
(3,6)	7		

Ecrire la formulation mathématique.

Exercice 22

Problème de routage de véhicules

Le représentant commercial de la brasserie (voir l'exemple 16) a conclu des contrats avantageux. Les pubs prendront des barils de dans les quantités indiquées dans le tableau suivant. Pour la livraison, un véhicule d'une capacité de 50 barils sera utilisé. L'objectif est de satisfaire toutes les exigences en minimisant la durée totale des circuits en véhicule.

	Exigence
1	0
2	18
3	10
4	15
5	12
6	10
7	8
8	11

Ecrire la formulation mathématique.

Exercice 23

1. Le problème SUBSET SUM est donné comme suit

INPUT: $n \in$ IN, n nombres naturels $a_1, \ldots, a_n$, nombre $B \in$ IN.

QUESTION: Existe-t-il un sous-ensemble $I \subseteq \{ 1, \ldots, n \}$ tel que $\sum_{i \in I} a_i = B$ *soit* vrai?

Le problème de partition PARTITION est donné comme suit

INPUT: $n \in$ IN, n nombres naturels $a_1, \ldots, a_n$.

QUESTION: Existe-t-il un sous-ensemble $I \subseteq \{ 1, \ldots, n \}$ tel que $\sum_{i \in I} a_i = \sum_{i / \in I} a_i$ est vrai?

(a) Montrer que SUBSET SUM est NP-complet. Vous pouvez supposer pour votre preuve qu'il est connu que PARTITION est NP-complete.

(b) Démontrez que PARTITION est NP-complète. Vous pouvez supposer pour votre preuve que SUBSET SUM est NP-complete.

2. Prouvez que le problème suivant est NP-difficile:

INPUT: $n \in$ IN, $n \times n$ matrice $D = (d_{ij})$.

QUESTION: Une tournée TSP d'une durée au plus (1+) fois la durée d'une tournée TSP la plus courte. (est une constante fixe > 0.)

Vous pouvez supposer que l'on sait que le problème du cycle hamiltonien (HC) est NP-complet.

3. Prouvez que le problème de clique maximum (CLIQUE) est NP-complet.

INPUT: Un graphe non orienté $G = (V, E)$, un nombre $k \in$ IN.

QUESTION: Existe-t-il une clique (sous-graphe complet de G) avec cardinalité (= nombre de sommets) $\geq k$?

Vous pouvez supposer que l'on sait que (VERTEX COVER) est NP-complet:

INPUT: Un graphe non orienté $G = (V, E)$ et un nombre $k \in$ IN.

QUESTION: Est-ce que G contient une couverture de sommet $C \subseteq V$ avec $\leq k$ sommets? (Un sous-ensemble V^0 de V est appelé couverture de vertex de G si chaque arête de E est incidente avec au moins un sommet de $V^{0.}$)

Exercice 24

Problème du facteur chinois non guidé

À l'Halloween, des enfants veulent visiter toutes les maisons du quartier (voir la figure). Les longueurs des rues (en mètres) qu'elles doivent traverser sont données dans le tableau. Allez-vous planifier une visite pour les enfants afin de minimiser la distance totale.

Arc	Longueur	Arc	Longueur
(1,2)	210	(6,7)	80
(1,9)	160	(6,11)	150
(2,3)	140	(7,8)	80
(2,5)	80	(7,9)	110
(3,4)	40	(9,10)	160
(3,5)	210	(10,11)	130
(4,6)	310	(10,12)	190
(5,6)	70	(11,12)	150

Ecrire la formulation mathématique.

Modéliser ce problème à l'aide d'un graphe et construire ce graphe.

CHAPITRE IV. PROGRAMMATION DYNAMIQUE

Plan

1. Introduction
2. Principe
3. Démarche
4. Exemples

1. Introduction

La programmation dynamique est un paradigme de conception d'algorithmes permettant de résoudre les problèmes (d'optimisation en particulier) pouvant être décomposés en sous problèmes qui chevauchent et qui satisfont un critère dit principe d'optimalité de Bellman : si le problème peut être formulé comme un graphe multiétage (où la solution est une suite de décisions) alors une portion d'une solution optimale est elle aussi optimale.

C'est une méthode exacte itérative considérée comme cas particulier du paradigme "diviser pour régner".

Pour résoudre un problème, on commence par résoudre les plus petits sous-problèmes et on conserve les valeurs de ces sous-problèmes dans une table de programmation dynamique. On utilise ensuite ces valeurs pour calculer la valeur de sous-problèmes de plus en plus grands, jusqu'à obtenir la solution de notre problème global. C'est une approche du bas vers le haut.

2. Principe

On décompose le problème µa résoudre en sous problèmes plus simples (similaires au problem initial).

On trouve, récursivement, la solution des sous problèmes (sauf si le problème est trivial, auquel cas on calcule directement la solution).

On combine les solutions des sous problèmes pour obtenir la solution du problème initial.

inventée par le mathématicien américain Richard Bellman dans les années 50 pour la résolution des problèmes d'optimisation;

Remarque

Programmation veut dire ici "planification".

3. Démarche

Etablir une équation de récurrence exprimant la solution d'un sous problème en fonction de celles de ses prédécesseurs (ou successeurs).

- Résoudre les sous problèmes récursivement ;
- Stocker leurs solutions dans une table
- Relever de cette table la solution du problème posé.

Question

Comment savoir si un problème d'optimisation peut être résolu par programmation dynamique ?

Réponse

Principe d'optimalité de Bellman : La solution optimale à un problème est composée de solutions optimales à des sous-problèmes.

4. Exemples de résolution par la PD

Exemple 1 : suite de Fibonacci

Définition

F(1) = 1; F(0) = 0 ; F(n) = F(n-1) + F(n-2).

n	0	1	2	3	4	5	6	7	8	9	…
fib (n)	0	1	1	2	3	5	8	13	21	34	…

Un algorithme itératif :

```
int fib(int n)
   (a, b) = (0, 1);
   For (i =2;i<n;i++)
      (a, b) =  (b, a + b);
   Return b;
```

Il est tout à fait évident que :

Complexité spatiale = O(1) ;

Complexité temporelle = O(n) ;

Un algorithme recursif

```
Int fib(int n)
If (n==0 // n==1) return n;
Return fib(n-1)+fib(n-2);
```

Ce n'est cependant pas une façon judicieuse de calculer la suite de Fibonacci, car on calcule de nombreuses fois les mêmes valeurs. Le temps de calcul est exponentiel en *n.*

F(n)

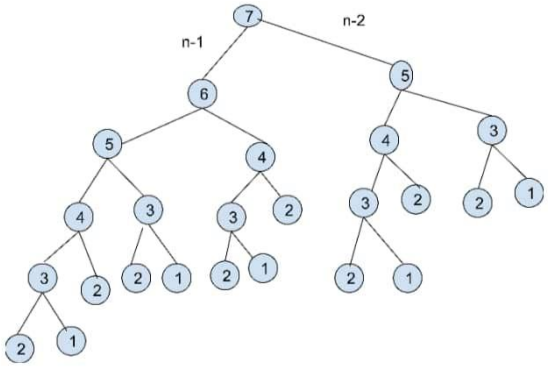
F(n-1) + F(n-2)

F(n-2) + F(n-3) F(n-3) + F(n-4)

F(n-3) + F(n-4) F(n-4) + F(n-5) F(n-4) + F(n-5) F(n-5)+F(n-6)

……

Arbre des appels pour n=7



Complexité

L'opération fondamentale est bien évidemment l'addiction. Soit T(n) le nombre d'Operations pour calculer le nième terme : T(0)= T(1)=0 et T(n)= T(n-1) + T(n-2) = O(φ^n) ou $\varphi = \frac{1+\sqrt{5}}{2}$.

Quiz : prouver ce résultat.

Résolution par la méthode de programmation dynamique :

On utilise une table F pour stocker les termes successifs de la suite :

F(0)	F(1)	F(2)	…						F(n-2)	F(n-1)	F(n)

Algorithme

```
int fib(int n)
   F (0)= 0 ;
   F (1)=1 ;
   For (i =2;i<n;i++)
      F (n)=  F (n-1) + F (n-2);
   Return b;
```

Il est clair que :

Complexité spatiale = O(n) ;

Complexité temporelle = O(n) ;

Exemple 2 : problème du rendu de monnaie

Enoncé

On a un montant M et n pièces de valeurs $v_1,v_2,...,v_n$. Trouver le nombre minimal de pièces à rendre dont la somme est M.

Soit x_i le nombre d'exemplaires de la pièce i de valeur v_i à rendre au client.

Le problème s'écrit alors :

$$\begin{cases} \min \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i v_i = M \\ x_i \in N \end{cases}$$

Solution

On va construire une table C[1..n,0..M]

Où C[i,j] = nombre minimum de pièces pour produire exactement le montant j en utilisant seulement des Pièces de valeurs $v_1, v_2, \ldots, v_i$.

La solution optimale sera donc en C[n,M].

On a donc les équations suivantes : C[i,0] = 0, i=1,n et C[i,j] = min(C[i-1,j],1 + C[i,j-vi])

Supposons qu'on ait un montant de 8 et 3 pièces de valeurs v1 = 1,v2 = 4 et v3 = 6

vi / j	0	1	2	3	4	5	6	7	8
v1 = 1	0	1	2	3	4	5	6	7	8
v2 = 4	0	1	2	3	1	2	3	4	2
v3 = 6	0	1	2	3	1	2	1	2	2

La solution optimale est (0,2,0).

Quiz : écrire l'algorithme correspondant et évaluer sa complexité.

Exemple 3 : calcul du coefficient binomial

Enoncé

Les coefficients binomiaux sont les coefficients du binôme :

$$(a + b)^n = C_n^0 a^n b^0 + \ldots + C_n^k a^k b^{n-k} + \ldots + C_n^n a^0 b^n$$

Récurrence: $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ pour n > k > 0 ; $C_n^k = 0$ pour k > n > 0

$C_n^0 = 1$, $C_n^n = 1$ pour n ≥ 0

On utilise une table C[0..n][0..k] (triangle de Pascal) :

n / k	0	1	2	…	j	…	k
0	1						
1	1	1					
.	1	2	1				
i					C(i,j)		
.							
n-1							
n							C(n,k)

Quiz : écrire l'algorithme correspondant et évaluer sa complexité.

Exemple 4 : problème du sac-à-dos

On dispose de n objets de poids positifs $w_1,w_2,...,w_n$ et de valeurs positives $v_1,v_2,...,v_n$. Notre sac à doc a une capacité en poids de W.

$$\begin{cases} max \sum_{i=1}^{n} x_i v_i \\ \sum_{i=1}^{n} x_i vw_i \leq W \\ x_i \in \{0,1\}, i = 1, n \end{cases}$$

On va construire une table de programmation dynamique V [1..n, 0..W]

où V [i,j] = valeur maximale des objets que l'on peut transporter si le poids maximal permis est j et que les objets que l'on peut inclure sont ceux numérotés de 1 à i.

La solution optimale sera en V [n,W]. On a donc les équations suivantes :

V [i,0] = 0, pour i=1,n et V [i,j] = max(V [i-1,j],V [i-1,j-wi] + vi)

Supposons qu'on ait 5 objets de poids 1,2,5,6 et 7 et de valeurs 1, 6, 18, 22, 28 et que la capacité de notre sac est de 11.

v_i	w_i		0	1	2	3	4	5	6	7	8	9	10	11
1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
6	2	2	0	1	6	7	7	7	7	7	7	7	7	7
18	5	3	0	1	6	7	7	18	19	24	25	25	25	25
22	6	4	0	1	6	7	7	18	22	23	28	29	29	40
28	7	5	0	1	6	7	7	18	22	28	29	34	35	40

L'optimum est alors 40 et la solution optimale est (0,0,3,4,0).

Quiz : écrire l'algorithme correspondant et évaluer sa complexité.

5. Exercices

Exercice 1

Triangle de Pascal

On veut calculer les coefficients binomiaux $C_n^k = \begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!}$.

Rappelons les propriétés suivantes :

$\begin{pmatrix} n \\ k \end{pmatrix} = \begin{pmatrix} n-1 \\ k-1 \end{pmatrix} + \begin{pmatrix} n-1 \\ k \end{pmatrix}$ pour $0 < k < n$, $\begin{pmatrix} n \\ n \end{pmatrix} = 1$ et $\begin{pmatrix} n \\ 0 \end{pmatrix} = 1$.

1. Donner un algorithme récursif du calcul de $\begin{pmatrix} n \\ k \end{pmatrix}$. Evaluer sa complexité.
2. Ecrire l'algorithme qui retourne $\begin{pmatrix} n \\ k \end{pmatrix}$ en utilisant la technique de la programmation dynamique. Evaluer sa complexité.

Exercice 2

Problème du stockage

Considérons n programmes $P_1,P_2,...,P_n$ qui peuvent être stockés sur un disque dur de capacité D gigabytes.

- Chaque programme P_i a besoin de s_i gigabytes pour être stocké et a une valeur v_i
- Tous les programmes ne peuvent pas être stockés sur le disque : $\sum_{i=1}^{n} s_i > D.$

Les programmes stockés dans le disque dur doivent maximiser la valeur totale, sans dépasser la capacité du disque dur. L'objectif est de concevoir un algorithme qui permet de calculer un tel ensemble.

Nous allons construire un tableau T dans lequel les lignes seront indexées par les programmes et les colonnes par les valeurs. L´élément $T[i,j]$ représentera la valeur maximale pour un disque dur de capacité j à l'aide des i premiers programmes.

1. Donner la formule de récurrence.
2. Donner l'algorithme utilisant la programmation dynamique.
3. Donner la complexité de cet algorithme.

Exercice 3

Problème de chemin le plus long dans un graphe

Soit $G = (V,E)$ un graphe orienté avec $V = \{v_1,...,v_n\}$. On dit que G est ordonné s'il vérifie les deux propriétés suivantes :

1. Chaque arc de ce graphe est de la forme $(i \rightarrow j)$ si $i < j$
2. Tous les sommets sauf le sommet v_n ont au moins un arc sortant.

Ici, par souci de simplification, nous supposerons qu'il existe un chemin allant de v_i vers v_n pour tout $i = 1,...,n$.

L'objectif est de trouver le chemin le plus long entre les sommets v_1 et v_n.

1. Montrer que l'algorithme glouton suivant ne résout pas correctement le problème.

   $u \leftarrow v_1$;
   $L \leftarrow 0$;
   Tant qu'il existe un arc sortant du sommet u
   choisir l'arc $(u \rightarrow v_j)$ tel que j est le plus petit possible
   $u \leftarrow v_j$;
   $L \leftarrow L + 1$;
   Retourner L
2. Donner la formule de récurrence qui permet de calculer la longueur du chemin le plus long commençant par v1 finissant par v`.
3. Donner un algo qui retourne la longueur du chemin le plus long commençant par v1 finissant par vn.
4. Modifier l'algorithme précédent afin qu'il retourne le chemin.
5. Donner la formule de récurrence permettant de calculer le chemin de poids maximum commençant par v1 finissant par vn.
6. En d´déduire l'algorithme.
7. Donner la formule de récurrence qui permet de calculer le chemin de poids maximal de ` arcs commençant par v1 finissant par vi . En déduire l'algorithme.

Exercice 4

Considérons un chef de projet qui doit gérer une équipe en lui affectant un projet qui dure une semaine. Le chef de projet doit choisir si il prend un projet stressant ou non stressant pour la semaine.

a) Si le chef de projet choisit le projet qui n'est pas stressant durant la semaine i, alors l'entreprise reçoit un revenu j.

b) Si le chef de projet choisit le projet qui est stressant durant la semaine i, alors l'entreprise recroit un revenu h_i et l'équipe ne travaille pas durant la semaine $i-1$

Exemple : Si chef de projet choisit de se reposer à la semaine 1, puis de prendre le projet stressant à la semaine 2, puis de prendre les projets non-stressant à la semaine 3 et 4. Le revenu total est de 70 et il correspond au maximum.

	semaine 1	semaine 2	semaine 3	semaine 4
l	10	1	10	1
h	5	50	5	1

1. Montrer que l'algorithme suivant ne résout pas correctement le problème.

pour chaque itération $i = 1,...,n$
  si $h_{i+1} > l_i + l_{i+1}$ alors
    Choisir Ne pas travailler à la semaine i
    Choisir le projet stressant à la semaine i+1
    Continuer à l'itération $i + 2$
  sinon
    Choisir le projet non-stressant à la semaine i
    Continuer l'itération $i + 1$

2. Donner un algorithme qui retourne le revenu maximal que peut obtenir le chef de projet.

Exercice 5

Multiplications chainées de matrices.

On veut calculer le produit de matrices $M = M_1M_2 \cdots M_n$. Multiplier une matrice $p \times q$, par une matrice $q \times r$ en utilisant la méthode standard nécessite pqr produit scalaire.

1. Considérons 4 matrices A : 20×5, B : 5×100, C : 100×8, D : 5×30. On veut calculer le produit $ABCD$. En fonction des parenthétisations, le nombre de produits varie.

D´déterminer le nombre de produits pour calculer $ABCD$, si on utilise les parenthétisations suivantes : $((AB)C)D$ ou $(A(BC))D$

2. Ecrire une formule de récurrence pour calculer $c(i,j)$.

3. Ecrire un algorithme utilisant la programmation dynamique

Exercice 6

1. Donner un algorithme de programmation dynamique pour résoudre le problème suivant :
   Entrée : une matrice A de taille $n \times m$ où les coefficients valent 0 ou 1.
   Sortie : la largeur maximum K d'un carré de 1 dans A, ainsi que les coordonnées (I, J) du coin en haut à gauche d'un tel carré (autrement dit pour tout i, j, $I \leq i \leq I + K - 1$, $J \leq j \leq J + K - 1$,
   $A[i, j] = 1$).
2. Quelle est sa complexité ?

Exercice 7

La bibliothèque planifie son déménagement. Elle comprend une collection de n livres $b_1, b_2, \ldots, b_n$. Le livre b_i est de largeur w_i et de hauteur h_i. Les livres doivent être rangés dans l'ordre donné (par valeur de i croissante) sur des étagères identiques de largeur L.

1. On suppose que tous les livres ont la même hauteur $h = h_i, 1 \leq i \leq n$. Montrer que l'algorithme glouton qui range les livres côte à côte tant que c'est possible minimise le nombre d'étagères utilisées.
2. Maintenant les livres ont des hauteurs différentes, mais la hauteur entre les étagères peut se régler. Le critère à minimiser est alors l'encombrement, défini comme la somme des hauteurs du plus grand livre de chaque étagère utilisée.

(a) Donner un exemple où l'algorithme glouton précédent n'est pas optimal.

(b) Proposer un algorithme optimal pour résoudre le problème, et donner son coût.

3. On revient au cas où tous les livres ont la même hauteur $h = h_i, 1 \leq i \leq n$. On veut désormais ranger les n livres sur k étagères de même longueur L à minimiser, où k est un paramètre du problème. Il s'agit donc de partitionner les n livres en k tranches, de telle sorte que la largeur de la plus large des k tranches soit la plus petite possible.

(a) Proposer un algorithme pour résoudre le problème, et donner son coût en fonction de n et k.

(b) On suppose maintenant que la la taille d'un livre est en $2^{o(kn)}$. Trouver un algorithme plus rapide que le précédent pour répondre à la même question.

Exercice 8

Étant donné un tableau T de n entiers relatifs, on cherche $\max\{\forall i, j \in \{1 \ldots n\} : \sum_{k=i}^{j} T[k]\}$.
Par exemple *pour* le tableau suivant :

2	18	−22	**20**	**8**	**−6**	**10**	−24	13	3

L'algorithme retournerait la somme des éléments 4 à 7 soit 32.

1. Donner un algorithme retournant la somme maximale d'éléments contigus par une approche *diviser pour régner*.

2. Donner un algorithme retournant la somme maximale d'éléments contigus par *programmation dynamique*.

3. Comparer la complexité Asymptotique au pire cas des deux approches.

CHAPITRE V. METHODE PAR SEPARATION ET EVALUATION

Plan

1. Introduction
2. Principe de l'algorithme
3. Domaines d'application
4. Exemple d'application
5. Exercices

1. Introduction

Pour plusieurs problèmes, en particulier les problèmes d'optimisation, l'ensemble de leurs solutions est fini (en tous les cas, il est dénombrable). Il est donc possible, en principe, d'énumérer toutes ces solutions, et ensuite de prendre celle qui nous arrange. L'inconvénient majeur de cette approche est le nombre prohibitif du nombre de solutions : il n'est guère évident d'effectuer cette énumération.

La technique du Branch & Bound est une méthode algorithmique classique pour résoudre un problème d'optimisation combinatoire. Il s'agit de rechercher une solution optimale dans un ensemble combinatoire de solutions possibles. La méthode repose d'abord sur la séparation (branch) de l'ensemble des solutions en sous-ensembles plus petits.

L'exploration de ces solutions utilise ensuite une évaluation optimiste pour majorer (bound) les sous-ensembles, ce qui permet de ne plus considérer que ceux susceptibles de contenir une solution potentiellement meilleure que la solution courante.

La méthode B&B a été proposée pour la première fois par Ailsa. H. Land and Alison. G. Doig en 1960 pour la programmation discrète.

En 2002 G. Gutin et A.P. Punnen publiaient un livre "The Traveling Salesman Problem and Its Variations". Ce livre couvre tous les domaines importants de l'étude sur TSP, y compris la théorie des polyèdres symétriques et asymétriques pour le TSP, branch and bound et d'autres méthodes.

2. Principe de l'algorithme

L'algorithme s'articule sur deux mécanismes :

Branch : Séparation.

Bound : Evaluation

Le branchement ou bien la séparation : diviser un ensemble de solutions en plusieurs sous-ensembles. La phase de séparation consiste à diviser le problème en un certain nombre de sous-problèmes qui ont chacun leur ensemble de solutions réalisables, de telle sorte que tous ces ensembles forment un recouvrement (idéalement une partition) de l'ensemble S. Ainsi, en résolvant tous les sous-problèmes et en prenant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial. Ce principe de séparation peut être appliqué de manière récursive à chacun des sous-ensembles de solutions obtenus, et ceci tant qu'il y a des ensembles contenant plusieurs

solutions. Les ensembles de solutions (et leurs sous-problèmes associés) ainsi construits ont une hiérarchie naturelle en arbre, souvent appelée arbre de recherche ou arbre de décision (Figure V.1.).

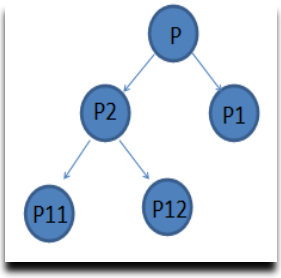


Figure V.1. Schéma de séparation.

L'évaluation d'un nœud de l'arbre de recherche a pour but de déterminer l'optimum de l'ensemble des solutions réalisables associé au nœud en question ou, au contraire, de prouver mathématiquement que cet ensemble ne contient pas de solution intéressante pour la résolution du problème (typiquement, qu'il n'y a pas de solution optimale). Lorsqu'un tel nœud est identifié dans l'arbre de recherche, il est donc inutile d'effectuer la séparation de son espace de solutions.

À un nœud donné, l'optimum du sous-problème peut être déterminé lorsque le sous-problème devient « suffisamment simple ». Par exemple, lorsque l'ensemble des solutions réalisables devient un singleton, le problème est effectivement simple : l'optimum est l'unique élément de l'ensemble. Dans d'autres cas, il arrive que par le jeu des séparations, on arrive à un sous-problème dans lequel les décisions « difficiles » ont été prises et qui peut ainsi être résolu en temps polynomial.

Pour déterminer qu'un ensemble de solutions réalisables ne contient pas de solution optimale, la méthode la plus générale consiste à déterminer un minorant du coût des solutions contenues dans l'ensemble (s'il s'agit d'un problème de minimisation). Si on arrive à trouver un minorant qui est supérieur au coût de la meilleure solution trouvée jusqu'à présent, on a alors l'assurance que le sous-ensemble ne contient pas l'optimum. Les techniques les plus classiques pour le calcul de minorants sont fondées sur l'idée de relaxation de certaines contraintes : relaxation continue, relaxation lagrangienne, etc.
La phase d'évaluation d'un nœud de l'arbre de décision a pour but de déterminer :

- Que cet ensemble ne contient pas de solution intéressante pour la résolution du problème (i.e. LB> min (coût (si))) donc pas de séparation
- De déterminer l'optimum de l'ensemble des solutions réalisable associé au nœud question ie: LB<min (coût(si)) et ceci pour un cas de minimisation de la fonction du coût (Figure V.2.)

Pour représenter une PSE, nous utilisons un « arbre de recherche »
Il y a deux façons pour explorer l'arbre :
- En profondeur : nécessite des retours arrière.
- Valeur minimale : le nœud dont l'évaluation est minimale.

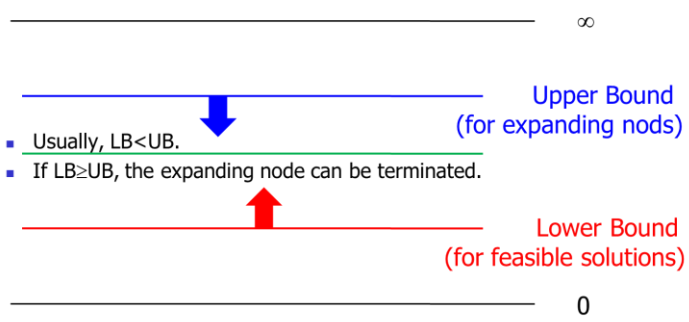


Figure V.2. Choix des bornes.

3. Algorithme général

```
New queue q of sol
enqueue (racine,q)
Nœud = racine
A= LB(noeud)
B= UB(noeud)
Solopt = noeud
Fopt=B      // Fopt= A en cas de maximisation)
While not empty(q)
        dequeue(q)
        If A=B
                Solopt = noeud
                Fopt = B
                Exit while
        Else
                If  A < B and A < Fopt
                        For each child c of noeud enqueue (c,q)
                        If B<Fopt  //  if (A> Fopt) en cas de max)
                                Fopt=B   // Fopt= A en cas de max
                                Solopt=noeud
                Endif
        Endif
        Noeud = peek(q)
        A= LB(noeud)
        B= UB(noeud)
End while
Print(fopt, solopt)
```

La technique abordée dans cette approche est une méthode exacte d'optimisation qui rend optimale la fonction objective et pratique une énumération intelligente de l'espace des solutions. Il s'agit en quelque sorte d'énumérations complètes améliorées. Elle partage l'espace des solutions en sous-ensembles de plus en plus petits et est appliquée à des problèmes NP-difficiles. Cette méthode reste bien sûr exponentielle, mais sa complexité en moyenne est bien plus faible que pour une énumération complète. Pour des problèmes de grande taille, sa durée d'exécution devient prohibitive, et il faut se tourner vers des heuristiques.

La méthode générale pour résoudre des problèmes d'optimisation discrète prend un temps exponentiel dans le pire des cas, utile pour résoudre de petites instances de problèmes difficiles

Peut être appliqué à tous les problèmes de planification
Il faut préciser :

- → Un schéma de borne inférieure
- → Règle de branchement (séparation).
- → Règle de recherche (règle de sélection de sous-problème)

4. Démarche générale

Pour un problème de minimisation, à une branche donnée et un nœud d'arbre lié (b&b) i :

1. Obtenir une borne inférieure, LBi
2. (Facultatif) Obtenez une solution candidate et un UBi de borne supérieure
3. Si (UBi < GLOBAL UB), alors définissez ubi comme GLOBAL UB
4. Si (LBi >= GLOBAL UB), alors élaguez le nœud et passez à 6.
5. Branchez-vous en sous-problèmes (créez des nœuds enfants)
6. Choisissez le sous-problème actif suivant.

S'il n'en existe pas, arrêtez. Sinon, passez à 1.

Exemple 1

Problème : Une seule machine, n tâches arrivant à des moments différents, préemptions non autorisées, minimisez le retard maximum (L_{max})
• Un nœud B&B correspond à un ordonnancement partiel
• Au niveau k de l'arbre B&B, les k premiers jobs du planning ont été corrigés
• Branchement
Créez un nœud enfant d'un nœud au niveau k-1 en fixant chaque tâche restante à la k-ième position SAUF :
Si une tâche c satisfait, ne créez pas d'enfant pour cette tâche.
• Borne inférieure :
Planifiez les tâches restantes selon la règle EDD préemptive :

Chaque fois que la machine est libérée ou qu'une nouvelle tâche est libéré, choisissez la tâche inachevée avec une date d'échéance minimale

Exemple numérique:

Jobs j	1	2	3	4
p_j	4	2	6	5
d_j	8	12	11	10
r_j	0	1	3	5

La borne inférieure au nœud (1,-) est obtenue en appliquant un EDD préemptif aux tâches 2, 3 et 4 avec l'heure de début 4.

At t=4,	available jobs: 2, 3,	pick 3
At t=5,	available jobs: 2, 3,4	pick 4
At t=10,	available jobs: 2, 3,	pick 3
At t=15,	available jobs: 2,	pick 2

L1 = -4, L2 = 5, L3 = 4, L4 = 0, Lmax = 5

Quiz : calculer LB au nœud (2,-).

EDD au nœud (1,3,-) fournit un ordonnancement non préemptif 1, 3, 4, 2 avec Lmax = 5 et c'est la solution optimale au probleme.

5. Exercices

Exercice 1
Coloration de Graphe : Étant donné un graphe, coloriez les sommets de telle sorte qu'aucun sommet adjacent n'ait la même couleur. Utilisez le plus petit nombre de couleurs possible.

Exercice 2
Satisfiabilité Maximale : Étant donné une expression booléenne dans CNF, trouvez une affectation de vérité aux littéraux qui satisfait autant de clauses que possible.

Exercice 3
Programmation linéaire entière : étant donné une fonction objectif f(x0, x1, ..., xn) et un ensemble de contraintes linéaires de la forme sum(ci * xi) <= k, recherchez l'ensemble de valeurs entières pour x0, x1, ..., xn qui satisfont aux contraintes pour trouver des valeurs qui maximisent f.

Exercice 4
Chemin le plus long : étant donné un graphique, trouvez le chemin le plus long (séquence d'arêtes qui touche chaque sommet au plus une fois).

Exercice 5
Ensemble indépendant maximum : étant donné un graphique, trouvez le plus grand ensemble S de sommets possible tel qu'aucun des sommets de S ne soit adjacent les uns aux autres.

Exercice 6
Clique maximale : étant donné un graphique, trouvez le plus grand ensemble S de sommets possible tel que tous les sommets de S soient adjacents les uns aux autres (ok, si vous avez fait le précédent, c'est assez trivial !). Vous avez peut-être remarqué que beaucoup d'entre eux sont des versions d'optimisation de problèmes NP-Complete classiques. Continuons sur ce thème.

Exercice 7
Correspondance de triangles : étant donné un graphique, trouvez le nombre maximum d'ensembles de sommets disjoints {a, b, c} tels que dans chaque ensemble, tous les sommets soient adjacents. Par DISJOINT (ce qui est naturellement plus facile à comprendre car je crie), nous entendons qu'aucun sommet ne peut appartenir à plus d'un des ensembles.

Exercice 8
Couverture minimale des sommets : étant donné un graphique, trouvez le plus petit ensemble S de sommets possible tel que chaque arête du graphique ait au moins une extrémité dans S.

Exercice 9
Problème de réparateur itinérant : étant donné un graphe G avec des arêtes pondérées et un sommet de départ s, trouver un parcours (pas nécessairement un chemin) du graphe qui commence à s, visite chaque sommet au moins une fois et minimise la somme des poids des arêtes traversées avant que chaque sommet ne soit atteint pour la première fois. (Si cela semble déroutant, l'histoire

derrière le titre du problème devrait vous aider - imaginez qu'un réparateur doive réparer une machine à chaque sommet et que les poids sur les bords représentent le temps nécessaire pour parcourir les bords. Le temps d'attente pour chaque le sommet est la somme des poids de toutes les arêtes parcourues avant que le sommet ne soit atteint pour la première fois. Nous voulons minimiser le temps d'attente total).

Exercice 10

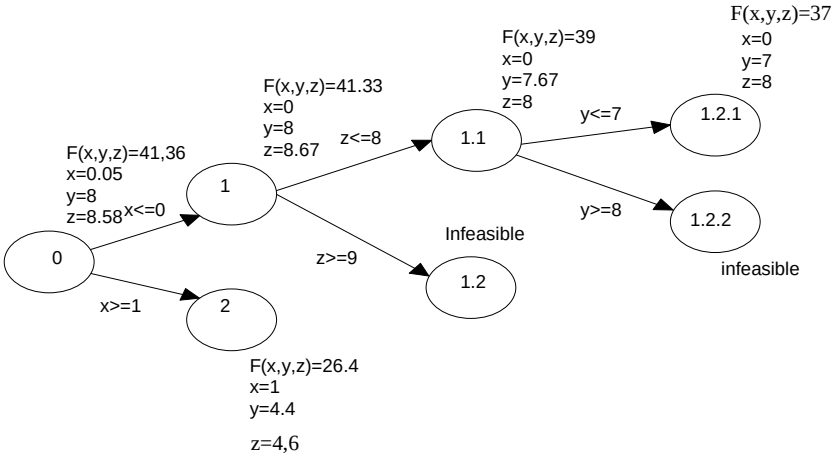
Problème du vendeur itinérant : étant donné un graphe complet G avec des arêtes pondérées avec un sommet de départ s et un sommet d'arrivée f, trouver un chemin qui commence à s, se termine à f, visite chaque sommet exactement une fois, et minimise le plus grand poids d'arête utilisé dans le chemin. (Encore une fois, une histoire peut aider : si les pondérations représentent le temps de trajet, nous souhaitons alors minimiser le temps le plus long que le vendeur passe sur la route entre les arrêts.)

Exercice 11

Partitionnement de graphe : étant donné un graphe G et un entier k, partitionnez les sommets de G en k ensembles de telle sorte que le nombre d'arêtes qui ont leurs extrémités dans différents ensembles soit minimisé. Ce problème a des applications dans le domaine de l'informatique distribuée. Si les sommets représentent des processus et les arêtes représentent des données partagées et que nous avons k processeurs dans le réseau, alors nous souhaiterons peut-être attribuer des processus aux processeurs de telle manière que chaque processeur soit occupé et que le partage de données entre processeurs soit minimisé.

Exercice 12

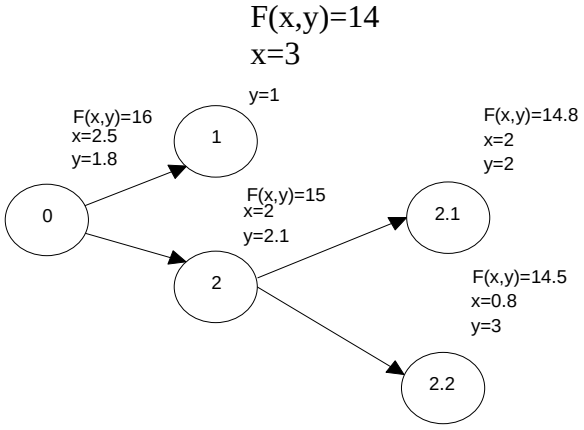
Étant donné un certain problème de programmation de ligne entière avec la branche et l'arbre lié suivants :



a) Déterminez si l'objectif du problème est de maximiser ou de minimiser.
b) Avons-nous atteint la solution optimale ? Si c'est le cas, expliquez pourquoi, sinon dites à partir de quel nœud nous devrions créer une branche et avec quelles contraintes.
c) Écrivez le problème résolu dans le nœud 1.2.1.

Exercice 13

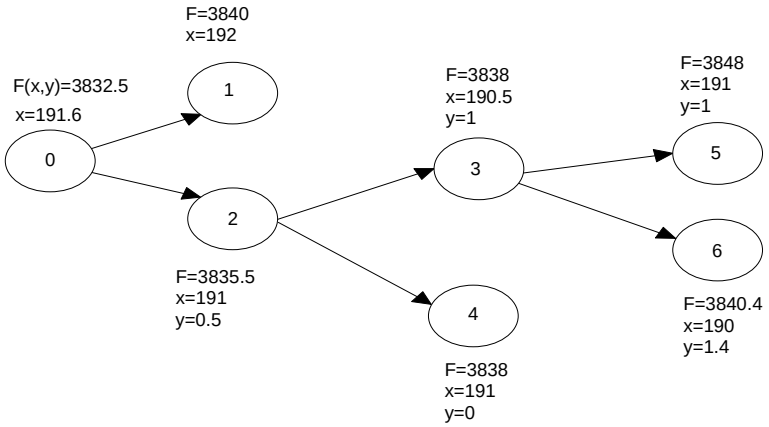
Étant donné un certain problème de programmation linéaire en nombres entiers avec l'arbre de branchement et de liaison suivant :



a) Écrivez la contrainte ajoutée à chaque branche.
b) Avons-nous atteint la solution optimale ? Si tel est le cas, expliquez pourquoi, sinon dites à partir de quel nœud nous devrions créer une branche et avec quelles contraintes.

Exercice 14

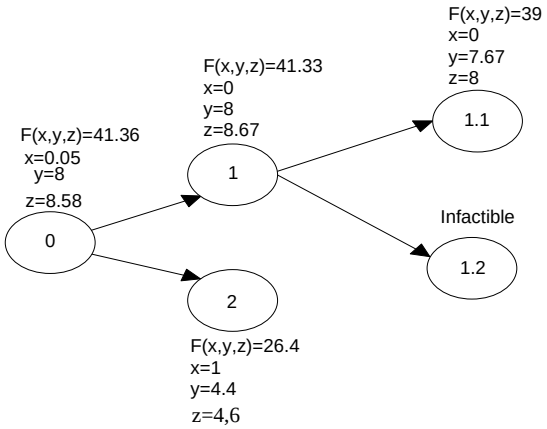
Considérons un problème de programmation linéaire en nombres entiers pour lequel les données suivantes de son arbre branché et lié sont connues :



a) Déterminez s'il s'agit d'un problème de maximisation ou de minimisation.
b) Avons-nous atteint la solution optimale ? Si c'est le cas, expliquez pourquoi, sinon dites à partir de quel nœud nous devrions créer une branche et avec quelles contraintes.
c) Écrivez le problème résolu dans le nœud 6.

Exercice 15

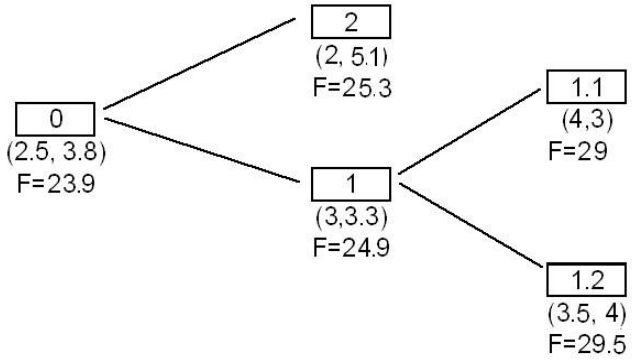
Étant donné un certain problème de programmation linéaire en nombres entiers mixtes, où les variables x et z ont une condition d'intégralité, avec l'arbre de branchement et de liaison suivant :



a) Déterminez s'il s'agit d'un problème de maximisation ou de minimisation.
b) Avons-nous atteint la solution optimale ? Si c'est le cas, expliquez pourquoi, sinon dites à partir de quel nœud nous devrions créer une branche et avec quelles contraintes.
c) Écrivez le problème résolu dans le nœud 1.2.

Exercice 16

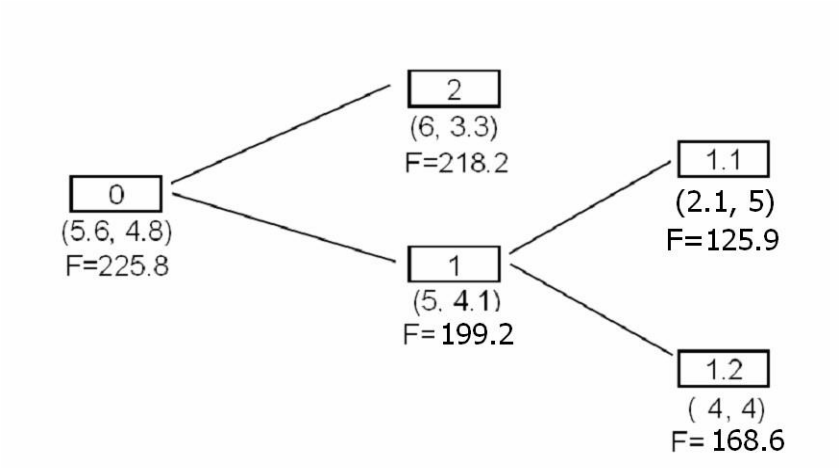
Considérons un problème ILP avec deux variables x et y. Supposons que nous ayons l'arbre de branchement et de liaison associé suivant :



a) Déterminez s'il s'agit d'un problème de maximisation ou de minimisation.
b) Écrivez les contraintes qui ont été ajoutées à chaque branche.
c) Déterminez si la solution optimale a déjà été trouvée. Sinon, effectuez une branche à partir du nœud approprié.
d) Si le problème initial était un problème ILP mixte où seule la variable x devait être entière, quelle serait la réponse à la question (c) ?

Exercice 17

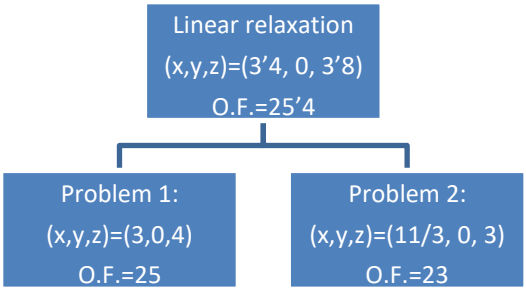
Considérons un problème ILP avec deux variables x et y. Supposons que nous ayons l'arbre de branchement et de liaison associé suivant :



a) Déterminez s'il s'agit d'un problème de maximisation ou de minimisation.
b) Écrivez les contraintes qui ont été ajoutées à chaque branche.
c) Déterminez si la solution optimale a déjà été trouvée. Sinon, effectuez une branche à partir du nœud approprié.
d) Si le problème initial était un problème ILP mixte où seule la variable x devait être entière, quelle serait la réponse à la question (c) ?

Exercice 18

Un problème linéaire avec trois variables entières est associé à l'arbre de branchement et de liaison suivant :



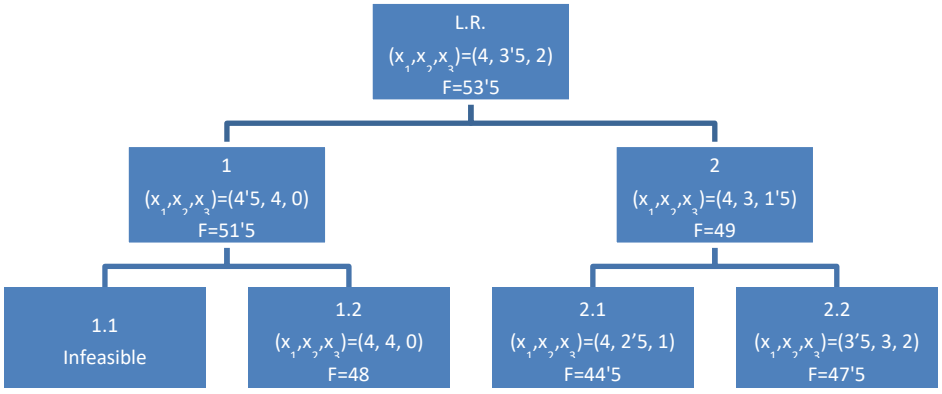
a) Écrivez la contrainte qui a été ajoutée à chaque branche.

Cochez la bonne option :
La solution optimale ne peut pas encore être obtenue : il faut bifurquer sur le nœud 2.

- La solution optimale est (3, 0, 4) avec OF=25
- La solution optimale est (11/3, 0, 3) avec OF=23
- La solution optimale est (4, 0, 4) avec OF =28

Exercice 19

Les solutions suivantes ont été obtenues dans un problème de programmation linéaire en nombres entiers :



a) Écrivez les contraintes qui ont été ajoutées à chaque branche.
b) Déterminez si la solution optimale a été trouvée. Si c'est le cas, dites de quoi il s'agit, sinon dites de quel problème nous devrions dériver et avec quelles contraintes.

Exercice 20

Considérons le problème de programmation en nombres entiers mixtes suivant :

$$Min\ .\ F = 3\,x_1 + 2\,x_2 + 5\,x_3$$
$$s\ .\ t\ .\ 2\,x_1 + x_2 + 2\,x_3 \geq 12$$
$$x_1 + x_2 + 4\,x_3 \geq 8$$
$$x_1, x_2, x_3 \geq 0$$
$$x_3 \text{ entier}$$

Si le premier problème résolu lors de l'application de la méthode branch-and-bound fournit la solution optimale (x_1, x_2, x_3) = (16 3,0, 2 3) , avec F = 58/3, cochez la case (es) qui ont du sens pour les solutions des sous-problèmes obtenus après le premier branchement :

- (x_1, x_2, x_3) = (6, 0,12) avec F =20'5 et (x_1, x_2, x_3) = (5,1,12) avec F =19'5
- (x_1, x_2, x_3) = (4,4,0) avec F =20 et (x_1, x_2, x_3) = (5, 0,1) avec F =20
- (x_1, x_2, x_3) = (6, 0, 0) avec F =18 et (x_1, x_2, x_3) = (4,1,1) avec F =19
- (x_1, x_2, x_3) = (3,4,1) avec F =22 et (x_1, x_2, x_3) = (6,1, 0) avec F =22

CHAPITRE VI. ALGORITHMES GLOUTONS

Plan

1. Introduction
2. Principe général
3. Définition
4. Exemples d'application
5. Exercices

1. Introduction

Les méthodes algorithmiques peuvent être regroupées en familles, parmi lesquelles figure celle des algorithmes dits gloutons. Nous allons dans un premier temps, définir ce qui caractérise un algorithme glouton, puis nous présenterons quelques algorithmes gloutons pour résoudre différents problèmes d'optimisation.

Les algorithmes gloutons sont des algorithmes assez simples dans leur logique. Ainsi que leur nom le suggère, ils sont conçus pour prendre le maximum de ce qui est disponible à un moment donné.

2. Principe général

L'algorithme glouton choisit la **solution optimale** qui se présente à chaque instant, sans se préoccuper, ni du passé ni de l'avenir. Il répète cette même stratégie à chaque étape jusqu'à avoir entièrement résolu le problème.

Il consiste à faire un choix localement optimal dans l'espoir que ce choix mènera à la solution globalement optimale.

Un algorithme glouton fonctionne parfois bien pour les problèmes d'optimisation. Cela fonctionne par phases.

A chaque phase :

On prend le meilleur dès maintenant, sans se soucier des conséquences futures.

On espére qu'en choisissant un optimum local à chaque étape, on aboutira à un optimum global.

3. Définition

Un algorithme glouton est un algorithme qui effectue à chaque instant, le meilleur choix possible sur le moment, sans retour en arrière ni anticipation des étapes suivantes, dans l'objectif d'atteindre au final un résultat optimal.

Remarque

Les algorithmes gloutons sont parfois appelés algorithmes gourmands ou encore algorithmes voraces.

La répétition de cette stratégie très simple, permet de résoudre rapidement et de manière souvent satisfaisante des problèmes d'optimisation sans avoir à tester systématiquement toutes les possibilités. Cependant un algorithme glouton ne fournit pas toujours le meilleur résultat possible.

À retenir

La solution obtenue par un algorithme glouton est le résultat d'une **suite de choix** gloutons, sans prise en compte des choix passés, ni anticipation de choix futurs.

L'optimisation est donc potentiellement moindre qu'un algorithme effectuant une exploration systématique de toutes les possibilités.
Toutefois, les algorithmes gloutons sont généralement **moins coûteux** qu'une exploration systématique. Ils sont ainsi capables de produire rapidement des résultats qui s'avèrent, dans de nombreux cas, suffisamment optimisés pour être acceptables.

Les algorithmes gloutons sont souvent employés pour résoudre des **problèmes d'optimisation combinatoire**.
Exemples
Les algorithmes gloutons se montrent efficaces pour :

- Déterminer le plus court chemin dans un réseau ;
- Optimiser la mise en cache de données ;
- Compresser des données ;
- Organiser au mieux le parcours d'un voyageur visitant un ensemble de villes ;
- Organiser au mieux des plannings d'activité ou d'occupations de salles.

Les algorithmes gloutons sont simples dans leur logique et donc faciles à implémenter, même si la détermination du choix glouton pertinent est plus ou moins évidente selon les cas.

4. Exemples d'application

Exemple 1

Rendu de monnaie (l'énoncé et la formulation mathématique ont été décrits au chapitre III)
Un **outil de rendu de monnaie** pourra être intégré à un automate distributeur ou à une caisse enregistreuse afin de proposer un rendu de monnaie, permettant de préserver le fonds de caisse en maximisant ainsi le revenu de l'activité (puisque si un certain type de pièce de monnaie est épuisé, alors pas de vente donc pas de revenu et vice versa). Le fonds de caisse est constitué d'un nombre limité de pièces de monnaie de banque servant à restituer le trop-perçu dans une transaction d'espèces.
Intuitivement, on préférera rendre une pièce de 100 dinars plutôt que 5 pièces de 20 dinars afin de conserver une monnaie suffisante dans notre fonds de caisse, et pouvoir ainsi continuer à rendre la monnaie le plus longtemps possible.

Stratégie gloutonne
Notre stratégie gloutonne consiste à rendre à chaque fois les pièces de plus grandes valeurs possibles, en plus grande quantité possible.
Nous devons donc déterminer pour tout montant quelle est la pièce de plus grande valeur qu'il est possible de rendre, et combien on peut en donner sans dépasser le montant restant à rendre.

Implémentation

Nous définissons notre monnaie, c'est-à-dire l'ensemble des valeurs faciales des pièces que nous pourrons utiliser pour notre fonds de caisse, triées par ordre croissant.
Monnaie = {1, 2, 5, 10, 20, 50, 100, 200} (Par simplification, on ne traite pas les centimes.)
Algorithme

```
Int rendu_monnaie(int montant, int[] monnaie):
r = 0    // r est une variable qui compte le nombre de pièces rendus
reste = montant // montant restant à rendre
for (i= len(monnaie)- 1, i>0, i--):
      {r += reste / monnaie[i] ; reste = reste % monnaie[i] ;}
return r
```

print(rendu_monnaie(48, monnaie)) // affiche 5

- → pièces de 20 dinars ;
- → pièce de 5 dinars ;
- → pièce de 2 dinars ;
- → 1 pièce d'1 dinar.

print(rendu_monnaie(81, monnaie)) // affiche 4

- → 1 pièce de 50 dinar ;
- → 1 pièce de 20 dinar ;
- → 1 pièce de 10 dinar ;
- → 1 pièce d'1 dinar.

Pour le système américain, l'algorithme glouton donne toujours la solution optimale, ce n'est le cas pour les autres systèmes.

Problème du sac à dos simple 0/1

(L'énoncé et la formulation mathématique ont été décrits au chapitre III)
Un algorithme glouton qui résout ce problème consiste à prendre les objets par ordre décroissant des rapport vi/wi des objets jusqu'au remplissage du sac.
Appliquons cet algorithme à l'instance suivante :

n = 5 objets ; capacite du sac W = 15					
Objet i	1	2	3	4	5
Valeur vi	5	4	2	5	3
Poids wi	2	8	3	5	9
Rapport vi/wi	2.5	0.5	0.66	1	0.33

Tri

Objet i	1	4	3	2	5
Rapport vi/wi	2.5	1	0.66	0.5	0.33
Poids cumulé	2	7	10	/	/
Valeur cumulée	5	9	11	/	/

La solution fournie par cet algorithme glouton est donc {1,4,3} avec une valeur cumulée de 14 et un poids cumulé de 15.
On peut aisément remarquer que cette solution n'est pas optimale. Une énumération exhaustive donne la solution optimale {1,2,4} avec une valeur cumulée de 11 et un poids cumulé de 10.

Remarque

Ce même algorithme pourrait fournir une solution optimale pour une autre instance.

Exemple 3

Compression de données : Codage de Huffman
Un codage de longueur variable est un codage optimal (économie de 25 % par rapport à un codage de longueur fixe).

L'algorithme de codage de Huffman est un algorithme glouton
Étant donné le pourcentage d'apparition de chaque caractère dans un corpus, déterminez un modèle de bits variable pour chaque caractère.
Vous choisissez toujours les deux plus petits pourcentages à combiner.
On considère qu'aucun mot de code n'est aussi préfixe d'un autre mot de code. Ce type de codage est appelé codage préfixe.
Le processus de décodage exige que le codage ait une représentation commode. On peut le représenter sous la forme d'un arbre binaire.

```
HUFFMAN(C)
n ← |C|
F ← C
pour i ← 1 à n-1 faire
        z ← ALLOUER_NOEUD()
        x ← gauche
        [z] ← EXTRAIRE_MIN(F)
        y ← droite[z] ← EXTRAIRE_MIN(F)
        f[z] ← f[x] + f[y]
        INSERER(F, z)
```

Quiz : appliquer cet algorithme à un exemple.

D'autres algorithmes gloutons

- L'algorithme de Dijkstra pour trouver le chemin le plus court dans un graphe.
  Prend toujours le bord le plus court reliant un nœud connu à un nœud inconnu
- L'algorithme de Kruskal pour trouver un arbre couvrant à coût minimum

Essaie toujours l'avantage restant le moins coûteux

- L'algorithme de Prim pour trouver un arbre couvrant à coût minimum
  Prend toujours l'avantage le moins coûteux entre les nœuds du spanning tree et les nœuds pas encore dans le Spanning Tree.

5. Exercices

Exercice 1

Un cambrioleur possède un sac à dos d'une contenance maximum de 30 kg. Au cours d'un de ses cambriolages, il a la possibilité de dérober 4 objets A, B, C et D. Voici un tableau qui résume les caractéristiques de ces objets :

Objet	**A**	**B**	**C**	**D**
Masse	13 kg	12 kg	8 kg	10 kg
Valeur marchande	700	400	300	300

On ajoute les contraintes suivantes :

- Le sac à dos a une contenance de 30 kg
- Le cambrioleur cherche à obtenir un gain maximum.

1. Déterminez les objets que le cambrioleur aura intérêt à dérober.
2. Quel critère pourrait-on choisir pour trier les objets ? Proposer un algorithme glouton pour résoudre le problème du sac à dos.
3. Retourne-t-il la solution optimale ?
4. Reprendre le problème avec les objets suivants :

Objet	**A**	**B**	**C**	**D**
Masse	35 kg	41 kg	28 kg	39 kg
Valeur marchande	70	40	30	30

5. a. Si le sac peut contenir 100 kg, b. Si le sac peut contenir 85 kg.
6. Écrire une fonction Python qui calcule les valeurs massiques d'une liste d'objets passés en paramètre.
7. Écrire une fonction Python qui renvoie le contenu d'un sac à dos depuis une liste d'objets et une contenance de sac à dos passés en paramètres. Cette fonction utilisera l'algorithme glouton présenté plus haut.

Exercice 2

On considère un jeu de pièce et une somme à rendre. Nous allons étudier deux situations générales, pour lesquelles l'algorithme glouton retourne la solution optimale ou non.

1. On considère un jeu de pièce similaire à l'euro : 1 cts, 2 cts, 5 cts, 10 cts, 20 cts, 50 cts, 1 €, 2 €, 5 €, 10 €, 20 €, 50 €, 100 €, 200 €.
   Rappeler l'algorithme glouton qui renvoie les pièces à rendre sous la forme d'une liste à partir d'un jeu de pièces et d'un montant passé en paramètres.
2. Donner les étapes pour le rendu de 71€73
3. Écrire une fonction Python qui traite le problème exposé à la question précédente.

Cette fonction retourne toujours une réponse mais celle-ci n'est pas forcement la meilleure.

3. Proposer un algorithme qui teste toutes les combinaisons possibles et renvoie la meilleure.
4. Comparer les complexités de deux algorithmes : glouton et exhaustif.

Exercice 3

On cherche à sélectionner cinq nombres de la liste suivante en maximisant leur somme et en s'interdisant de choisir deux nombres voisins.

Ainsi, si la liste contient la séquence 1, 2, 3, et qu'on choisit 2, il ne sera plus possible de choisir 1 ou 3.

Liste = 15, 4, 20, 17, 11, 8, 11, 16, 7, 14, 2, 7, 5, 17, 19, 18, 4, 5, 13, 8

1. Décrire une stratégie gloutonne pour résoudre ce problème.
2. Mettre en oeuvre cette stratégie sur cet exemple.
3. Vérifier que 20, 18, 17, 16, 15 est une solution acceptable.
4. Comparer ces solutions.

Exercice 4

Vous visitez un parc d'attractions proposant des spectacles à différents horaires. Voici les horaires des spectacles :

Spectacle	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
Horaires	10h-11h	10h30-11h30	11h-12h30	11h30-12h	12h-13h	13h-15h	13h30-14h	14h-15h30	15h-16h	16h-17h30

Vous avez remarqué qu'il n'était pas possible d'assister à tous les spectacles puisque certains ont lieu à des moments communs ; Vous souhaitez assister à un maximum de spectacles. Quels spectacles choisir ?

Voici deux stratégies gloutonnes possibles :

- **Stratégie 1** : choisir le spectacle dont l'heure de début arrive le plus tôt parmi les spectacles dont l'heure de début est postérieure à l'heure de fin des spectacles déjà choisis. Cette stratégie minimise le temps d'attente.
- **Stratégie 2** : choisir le spectacle dont l'heure de fin arrive le plus tôt parmi les spectacles dont l'heure de début est postérieure à l'heure de fin des spectacles déjà choisis. Cette stratégie maximise le temps restant aux autres spectacles.

1. Appliquer ces deux stratégies aux données.
2. Laquelle est la meilleure ?

Exercice 5

Un algorithme glouton pour colorier un graphe :

Prendre les sommets dans un ordre quelconque

Attribuer à chaque sommet la plus petite couleur non utilisée par ses voisins déjà coloriés

Combien de couleurs nécessite cet algorithme ? Est-il optimal ?

Exercice 6

Un algorithme glouton un peu plus malin pour colorier un graphe :

Prendre les sommets par degré décroissant

Attribuer à chaque sommet la plus petite couleur non utilisée par ses voisins déjà coloriés Que penser de cet algorithme par rapport au précédent ? Est-il optimal ?

Exercice 7

On se donne n réels {x1, x2, . . . , xn}. On souhaite trouver le nombre minimal K tel que K intervalles I1, I2, . . . IK chacun de longueur 1 recouvrent tous les points, c'est-à-dire que tout point appartient à au moins l'un de ces intervalles. On considère ici des intervalles fermés (dont les bornes sont incluses).

1. Donner un exemple (avec n petit) montrant qu'il peut y avoir plusieurs recouvrements minimaux (plusieurs familles d'intervalles pour le même K minimal). Donner également un exemple où ce recouvrement minimal est unique.
2. Écrire un algorithme glouton qui détermine la valeur minimale de K en construisant les K intervalles. 3. Écrire la preuve de votre algorithme glouton.

Exercice 8

Dans un petit restaurant, un serveur (seul) doit servir n clients ; il sait qu'il lui faudra ti minutes pour servir le client numéro i.

Le mécontentement de chaque client se mesure au temps qu'il doit attendre avant d'être servi. An de créer le moins de mécontentement possible, l'objectif est ici de minimiser le total des temps d'attente de tous les clients. Comment faut-il procéder et pourquoi ?

Exercice 9

Comment rendre une somme donné avec le minimum de pièces. Nous regarderons tout d'abord notre système monétaire : des pièces de 1, 2, 5, 10, 50, 100, 200. Q3.1 Comment rendre 263 centimes d'euros ? Q3.2 Trouvez un algorithme glouton pour ce problème. Q3.3 Montrez que cet algorithme est optimal. Q3.4 Qu'est ce qui se passe si les pièces ont comme valeur 1, 3, 4 ? Comment rendre la somme de 6 ?

Exercice 10

On cherche à sélectionner cinq nombres de la liste suivante en cherchant à avoir leur somme la plus grande possible (maximiser une grandeur) et en s'interdisant de choisir deux nombres voisins (contrainte). 15 - 4 - 20 - 17 - 11 - 8 - 11 - 16 - 7 - 14 - 2 - 7 - 5 - 17 - 19 - 18 - 4 - 5 - 13 - 8 Comme on souhaite avoir le plus grand résultat final, la stratégie gloutonne consiste à choisir à chaque étape le plus grand nombre possible dans les choix restants. 1. Appliquez cet algorithme glouton sur le tableau. 2. Vérifiez que est une autre solution possible. 3. Que dire de la solution gloutonne ?

Références bibliographiques

[1] G. Colson, Chr. De Bruyn. Models and methods in multiple criteria decision making, Pergamon, Oxford, 1989.

[2] K. Miettinen. On the methodology of multiobjective optimization with applications. Report 60, University of Jyvaskyla, Departement of Mathematics, Jvvaskyla, 1994.

[3] R.L. Keeney, H. Raiffa. Decision with multiple objectives: preferences and values trade-offs. Wiley, 1976.

[4] L.Y. Maystre, J. Pictet, J. Simos. Méthodes multicritères ELECTRE. Presses polytechniques et universitaires romandes, 1994.

[5] B. Roy, D. Bouyssou. Aide multicritère à la décision : méthodes et cas", Economica, 1993.

[6] J.C. Pomerol and S. Barba-Romero. Multicriterion decision in management: principles and practice, Kluwer Academic Publishers, 2000.

[7] P. Vallin , D. Vanderpooten. Aide à la decision. Une approche par les cas. Ed. Ellipses, Paris, 2002.

[8] Richard M. Karp, « Reducibility Among Combinatorial Problems », dans Raymond E. Miller et James W. Thatcher, Complexity of Computer Computations, Plenum, 1972 (ISBN 978-1-4684-2003-6, DOI 10.1007/978-1-4684-2001-2_9,), p. 85-103.

[9] A. H. Land et A. G. Doig, « An Automatic Method of Solving Discrete Programming Problems », Econometrica, vol. 28, no 3, 1960, p. 497–520 (ISSN 0012-9682, DOI 10.2307/1910129, lire en ligne [archive], consulté le 16 septembre 2020)

[10] Djamal Rebaïne, « Note de cours : La méthode de branch and bound » [archive], sur Université du Québec à Chicoutimi.

[11] Jens Clausen, « Branch and Bound Algorithms - Principles and Examples. » [archive], 1999.

[12] Stephen Boyd, and Mattingley Jacob. « Branch and Bound Methods », mars 11, 2007. Stanford University.

[13] Marcel, Turkensteen. « Advanced Analysis of Branch and Bound Algorithms » (2006). Netherlands.

[14] « Resolution par separation et evaluation: Branch & Bound », Ensimag 2A -2011 2010.

[15] « Séparation et évaluation ». Wikipédia, novembre 20, 2023. http://fr.wikipedia.org/w/index.php?title=S%C3%A9paration_et_%C3%A9valuation&oldid=84541985.