Paradigmes algorithmiques

Quelques méthodes de conception d'algorithmes

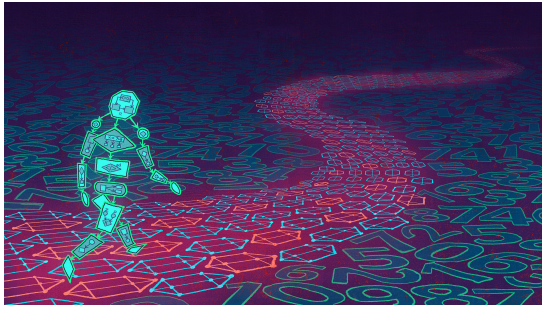
Gauthier Picard (Roland Jégou)

MINES Saint-Étienne

Introduction

Le processus général de résolution algorithmique (ou informatique) d'un problème peut grossièrement se décomposer en trois phases :

LE PROBLÈME $\rightarrow$ L'ALGORITHME $\rightarrow$ LE PROGRAMME



Introduction (suite.)

Le problème

- Préciser les spécifications : données ? résultats ? environnement ?
- Modélisation : structures mathématiques ? propriétés ? problème déjà connu, ... ?
- Modularisation : décomposition en sous problèmes, sous problèmes plus simples, indépendants, ... ?

Introduction (suite.)

L'algorithme

- Conception : méthodes de conception d'algorithmes ?
- Écriture dans un pseudo langage ?
- Structures de données ?
- Analyse : preuve (fin, validité), efficacité (complexités) ?

Introduction (suite.)

Le programme

- Traduction de l'algorithme dans un langage de programmation précis dans un environnement précis ?
- Mise au point : tests ? documentation ? ...

Introduction (suite.)

En pratique

- nombreux allers-retours entre ces étapes
- minimiser par une conception descendante
- du général au particulier, par l'utilisation de la modularisation
  - e.g. division en sous problèmes bien spécifiés et dont les dépendances doivent être clairement mises en évidence (sorte de graphe de précédence des différents « morceaux » : sous problèmes, modules, sous programmes, fonctions, ...)

Questions d'algorithmique

Outils de modélisations ?

- De nombreuses structures mathématiques sont très utiles (domaine des mathématiques discrètes, de la combinatoire) :
  - ensembles « particuliers » (suites, piles, files, tableaux, matrices, ...)
  - arborescences (structures fondamentales en informatique)
  - graphes (orientés ou non)
  - multigraphes
  - ensembles ordonnés
  - hypergraphes
  - ...
- Nécessité de connaître ces « objets », leurs propriétés et leurs mises en œuvre

Questions d'algorithmique (suite.)

Conception d'algorithmes ?

- Pas de recette miracle mais quelques méthodes générales de résolution
- Elles ne fonctionnent pas toujours mais elles ont le mérite de donner des pistes et de souvent fournir une solution
- On parle de Paradigmes Algorithmiques

Questions d'algorithmique (suite.)

Comparaison d'algorithmes ?

- Outil privilégié = Complexité (en fait *les* complexités, voir suite)
  - Donne une mesure objective de l'efficacité d'un algorithme
- Nécessité de maîtriser les calculs de complexités et de connaître certains résultats

Questions d'algorithmique (suite.)

Trouver de meilleurs algorithmes ?

- Essai de plusieurs stratégies algorithmiques et de différentes structures de données
- Comment déterminer la difficulté intrinsèque d'un problème ?
- Calculs de bornes inférieures ?
- Algorithmes optimaux ?
- Problèmes difficiles : Théorie de la Complexité (classes P, NP ; NP-complétude ; NP-difficulté ...)

Contenu de ce cours

Recherche Exhaustive

Elle consiste à examiner directement ou indirectement l'ensemble de toutes les solutions possibles (solutions admissibles ou réalisables)

Divide-and-Conquer, Récursivité

On résout le problème, en général récursivement, en le décomposant en sous problèmes de même nature, et en « fusionnant » les sous solutions obtenues

Prétraitement

Il s'agit ici d'organiser, de structurer des données pour mieux résoudre le problème

Méthodes incrémentales

Ces méthodes construisent la solution en considérant les éléments de la donnée les uns après les autres

Transformation, Réduction

L'idée est de ramener le problème à résoudre à un autre problème déjà connu. Cela permet aussi d'établir des résultats sur des bornes inférieures

Contenu de ce cours (suite.)

Programmation Dynamique

Le problème initial est décomposé en sous problèmes qui sont résolus de façon incrémentales, et non récursives, par tailles croissantes

Algorithmes Gloutons

Le problème est résolu de façon incrémentale en faisant à chaque étape un choix local qui n'est jamais remis en question

Menu

Menu

Recherche exhaustive

Appellée aussi Méthode Naïve (ou Algorithme Naïf)
en anglais « *Brute Force Method* », « *Naive Algorithm* »

Principe

Examiner tous les cas possibles

- Quoi de plus simple ?
- Encore faut-il qu'il y ait un nombre fini de cas possibles (toujours vrai ici)
- Dans ce cas, cette méthode donne toujours une solution en temps polynomial ou exponentiel (suivant le nombre de cas, d'où l'intérêt de savoir « compter »)
- Mais c'est une méthode pas toujours facile à mettre en œuvre (i.e. programmer)

C'est la méthode de base de résolution des Problèmes d'Optimisation Combinatoire (POC), car à la base des Méthodes Arborescentes (e.g. *branch-and-bound*)

Problèmes d'optimisation combinatoire

Définition 2.1 (POC)

Donnée S ensemble fini et f une fonction de S vers les entiers, $f : S \to \mathbb{N}$

Question On cherche $s_0 \in S$ telle que $f(s_0) = opt\{f(s), s \in S\}$ où opt signifie min ou max

Autrement dit, on cherche dans un ensemble fini S (ensemble des Solutions Réalisables ou Admissibles) un élément s_0 optimisant une certaine fonction f dite Fonction Objectif ou Economique

Problèmes d'optimisation combinatoire (suite.)

Souvent il existe un ensemble fini E sous-jacent et une fonction c associant un coût (profit, distance, poids, profit, capacité, ...) à tout élément de E tels que :

- $S \subseteq \mathfrak{P}(E)$ (S = famille de parties de E)
  i.e. $S = \{s \subseteq E, s \text{ vérifie une certaine propriété } \pi\}$
- $f(s) = \sum_{e \in S} c(e), \forall s \in S$

Le problème est alors dit Problème d'Optimisation Combinatoire à Fonction Objectif Séparée (POC-FOS)

Propriétés de ces ensembles

On note $\mathfrak{P}(E)$ l'ensemble des parties de E et $\mathfrak{P}_k(E)$ l'ensemble des parties de E ayant k éléments ($0 \leq k \leq n$)

$$|\mathfrak{P}(E)| = 2^n \tag{1}$$

$$|\mathfrak{P}(E)| = C_n^k = \binom{k}{n} = \frac{n!}{k!(n-k)!} \in \theta(n^k) \tag{2}$$

Le nombre de suites ordonnées de k éléments de distincts de E (arrangements) est

$$A_n^k = n \times (n-1) \times \ldots \times (n-k+1) = \frac{n!}{(n-k)!} \tag{3}$$

Si F est un ensemble ayant m éléments, alors le nombre d'applications de E dans F est égal à m^n (l'ensemble $A(E,F)$ des applications de E dans F est aussi noté F^E)

Génération exhaustive

- Ainsi pour un POC-FOS il est toujours possible de résoudre le problème par Génération Exhaustive = génération arborescente de tous les cas possibles : il y a en effet 2^n solutions potentielles à générer, quand E a n éléments
- On peut en effet associer à chaque élément i une variable booléenne x_i, valant 1 si l'élément i est choisi et 0 sinon
- Mais, même si elle est efficace quand n est petit, cela s'avère vite irréaliste par rapport aux temps de calcul, lorsque les tailles des données à traiter sont grandes

Exemple 2.2

Si $n = 100$, quelle serait la durée de la méthode sur une machine pouvant examiner chaque solution potentielle en 10^{-9} seconde ?

Génération exhaustive

- Ainsi pour un POC-FOS il est toujours possible de résoudre le problème par Génération Exhaustive = génération arborescente de tous les cas possibles : il y a en effet 2^n solutions potentielles à générer, quand E a n éléments
- On peut en effet associer à chaque élément i une variable booléenne x_i, valant 1 si l'élément i est choisi et 0 sinon
- Mais, même si elle est efficace quand n est petit, cela s'avère vite irréaliste par rapport aux temps de calcul, lorsque les tailles des données à traiter sont grandes

Exemple 2.2

Si $n = 100$, quelle serait la durée de la méthode sur une machine pouvant examiner chaque solution potentielle en 10^{-9} seconde ?

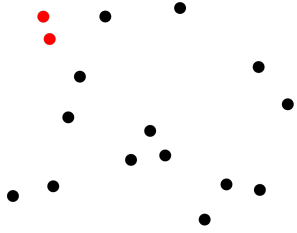
- $2^{100} \approx 1.25 \times 10^{30}$ solutions
- Temps = $1.26 \times 10^{30} \times 10^{-9} \approx 1.26 \times 10^{21}$ secondes $\approx 2.11 \times 10^{19}$ minutes $\approx 3.52 \times 10^{17}$ heures $\approx 1.46 \times 10^{16}$ jours $\approx 4 \times 10^{13}$ ans

Exemples de problèmes

Définition 2.3 (Paire de Points la Plus Proche (P^4))

Données Soit $E = \{p_1, p_2, \ldots, p_n\}$ un ensemble de n points du plan, chacun étant donné par ses coordonnées $p_i = (x_i, y_i)$

Question Trouver dans E deux points dont la distance est minimale



Exemples de problèmes (suite.)

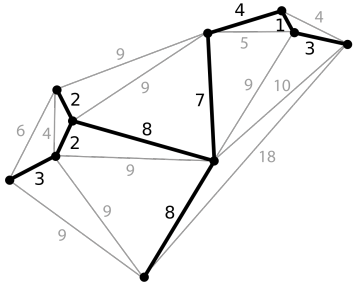
Définition 2.4 (Arbre Recouvrant Minimal (ou Minimal Spanning Tree))

Données Soit $G = (X, E, d)$ un graphe simple non orienté valué où d est une fonction qui attribue une valeur à chacune des arêtes (distance, coût, capacité, temps, ...)

Question Construire un arbre recouvrant $T = (X, E)$ de G qui soit de coût (somme des valeurs des arêtes) minimal

Théorème 2.5 (Cayley, 1897)

Il existe n^{n-2} arbres ayant pour sommets n points distincts du plan (ou pour le graphe complet K_n)

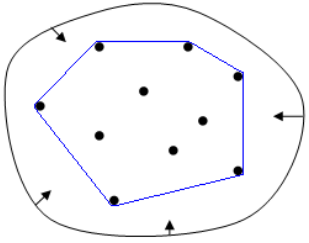


Exemples de problèmes (suite.)

Définition 2.6 (Enveloppe Convexe de n points du plan (EC))

Données Soit $E = \{p_1, p_2, \ldots, p_n\}$ un ensemble de n points du plan, chacun étant donné par ses coordonnées $p_i = (x_i, y_i)$

Question Déterminer l'Enveloppe Convexe de E, $EC(E)$, c'est-à-dire le plus petit ensemble convexe du plan contenant E

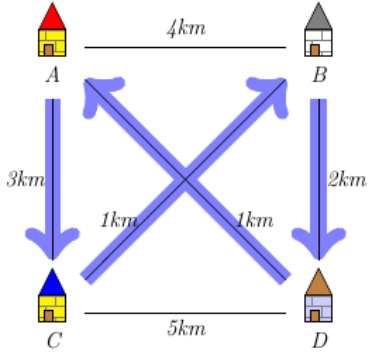


Exemples de problèmes (suite.)

Définition 2.7 (Voyageur de Commerce (PVC))

Données $G = (X, U, d)$ un graphe orienté valué où d attribue à chaque arc u une valuation $d(u)$ (coût, distance, temps, ...)

Question Déterminer un circuit hamiltonien de G de longueur minimale



Théorème 2.8

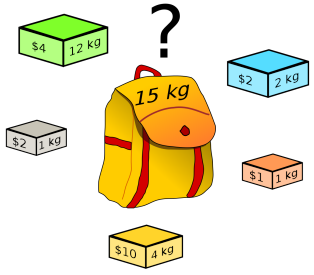
Il y a $1/2 \times (n-1)!$ *cycles possibles dans un graphe dans le plan*

Exemples de problèmes (suite.)

Définition 2.9 (Problème du Sac à Dos (SAD))

Données n objets, numérotés $1, 2, \ldots, n$ et d'utilités et poids respectifs $u_1, u_2, \ldots, u_n$ et $p_1, p_2, \ldots, p_n$ (entiers positifs) et d'un sac à dos de poids maximal P

Question Comment remplir le sac en prenant au plus un exemplaire de chaque objet et sans dépasser le poids maximal P pour maximiser son utilité ?



Exemples de problèmes (suite.)

Définition 2.10 (Problème de la Somme (SOM))

Données Soit $V = \{v_1, v_2, \ldots, v_n\}$ un ensemble d'entiers strictement positifs (valeurs, poids, durées, ...) et S un entier, avec $v_1 \leq v_2 \leq \ldots \leq v_n$

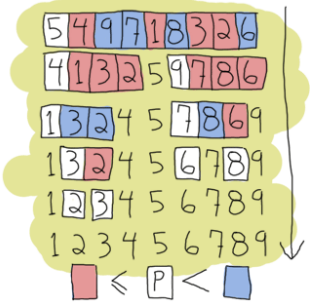
Question Existe-t-il un sous-ensemble W de V dont la somme des éléments est égale à S ou s'en approche de façon optimale ?

Exemples de problèmes (suite.)

Définition 2.11 (Le Problème du Tri (TRI))

Données Soit $E = \{e_1, e_2, \ldots, e_n\}$ un ensemble de n éléments muni d'un ordre total « $\leq$ », c'est-à-dire $\forall i, j$ on a soit $e_i \leq e_j$ soit $e_j \leq e_i$

Question Ordonner totalement les éléments de E



Méthode Branch-and-Bound

La méthode SEP (Séparation et Evaluation Progressive ou *Branch-and-Bound*, ou BB) est basée sur la Recherche Exhaustive mais permet d'atténuer l'explosion combinatoire

Principe

Parcourir implicitement et de façon arborescente (séparation), l'ensemble des solutions potentielles en tirant parti :

- *d'un ordre de sélection des variables*
- *d'une compatibilité des variables (le choix d'une variable peut éliminer ou au contraire impliquer d'autres variables)*
- *du calcul d'une fonction à chaque étape (représentant une borne inférieure pour un problème de minimisation ou un borne supérieure pour un problème de maximisation) permettant d'orienter la recherche et d'éliminer des recherches infructueuses (évaluation)*

Méthode Branch-and-Bound (suite.)

Borne inférieure

- Pour un problème de Maximisation il faut calculer une Borne Supérieure, $BS(x)$, en chaque sommet x de l' arborescence de l' exploration telle que toutes les solutions de la sous arborescence correspondante, $A(x)$, seront au plus égales à $BS(x)$
- Si $BS(x)$ est inférieure à un maximum déjà trouvé alors il est inutile d'explorer $A(x)$

Borne supérieure

- Pour un problème de Minimisation il faut calculer une Borne Inférieure, $BI(x)$, en chaque sommet x de l'arborescence de l'exploration telle que toutes les solutions de la sous arborescence correspondante, $A(x)$, seront au moins égales à $BI(x)$
- Si $BI(x)$ est supérieure à un minimum déjà trouvé alors il est inutile d'explorer $A(x)$

Méthode Branch-and-Bound (suite.)

Dans les deux cas, la recherche reprend au sommet le plus prometteur c'est-à-dire où la borne, $BS(x)$ ou $BI(x)$, est la meilleure, soit la plus grande pour un problème de Maximisation soit un le plus petite pour un problème de Minimisation

Branch-and-Bound

Représentation algorithmique

```
1  pb ← null
2  best ← +∞
3  Function branch_and_bound(pb_0)
4      if is_trivial(pb_0) then
5          local ← trivial_solve(pb_0)
6          if local < best then
7              best ← local
8              pb ← pb_0
9      else
10         local ← BI(pb_0)
11         if local < best then
12             (pb_1, pb_2) ← branch(pb_0)
13             branch_and_bound(pb_1)
14             branch_and_bound(pb_2)
```

Exemple d'utilisation de Branch-and-Bound

SAD

Exemple 2.12 (SAD)

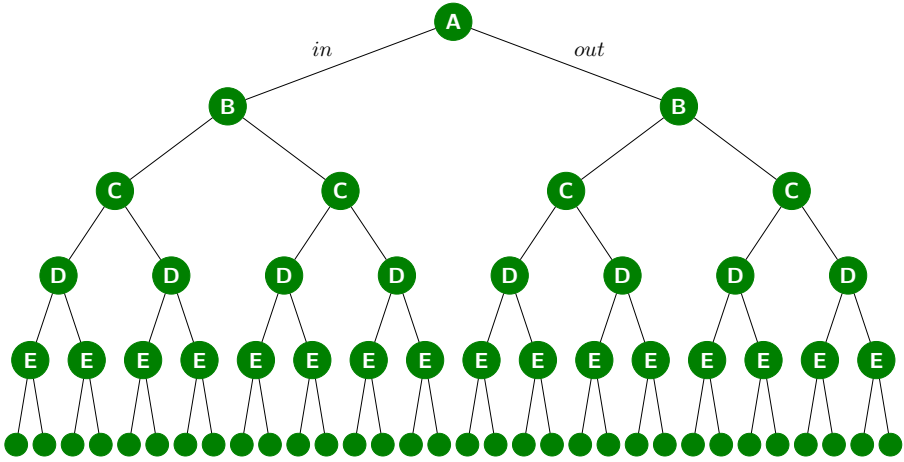
Comment remplir au mieux un sac de quantité max $P = 10$ avec les objets suivants :

	A	B	C	D	E
p_i	2	3.14	1.98	5	3
u_i	40	50	100	95	30

Exemple d'utilisation de Branch-and-Bound

SAD

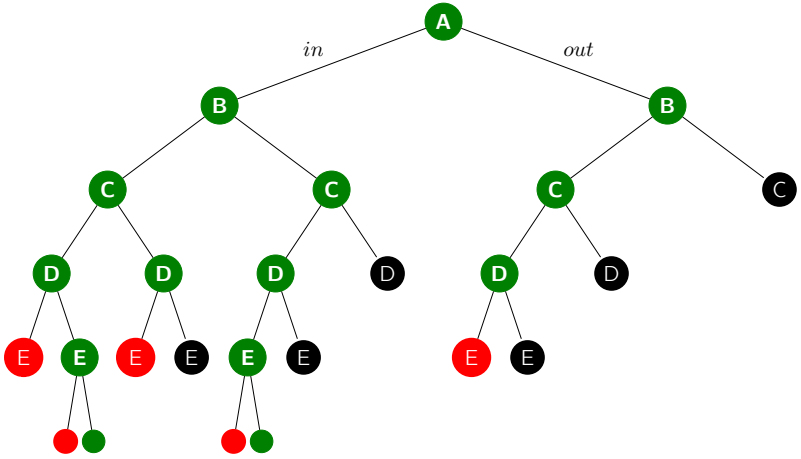
Exemple 2.13 (Résolution exhaustive)



Exemple d'utilisation de Branch-and-Bound

SAD

Exemple 2.14 (Branch-and Bound)



Conclusions sur Branch-and-Bound

- Simplicité de conception (essai potentiel de tous les cas possibles) mais mise en œuvre parfois difficile en ce qui concerne la génération effective de tous les cas et surtout le choix de la fonction permettant de calculer une borne
- Permet toujours d'obtenir une solution, mais au dépend du temps de calcul voire de l'espace mémoire
- C'est une des seules approches connue, avec la Programmation Dynamique, pour résoudre exactement les problèmes NP-Difficiles pour lesquels aucune autre stratégie est connue pour obtenir, plus efficacement, une solution optimale
- En général en Temps Exponentiel dans le pire cas pour les problèmes NP-Difficiles

La difficulté essentielle consiste donc à trouver un bon compromis «local-global » c'est-à-dire entre le temps global de calcul d'une solution et le temps local de calcul de la fonction permettant d'accélérer la recherche : plus la fonction sera élaborée plus elle élaguera l'arbre mais plus elle sera longue à calculer

Menu

Divide-and-Conquer

Principe

Le problème est décomposé en sous-problèmes de même nature et de tailles inférieures à celle du problème dont les solutions sont calculées par la même méthode, donc récursivement. La solution globale est alors obtenue par recomposition ou fusion des solutions des sous-problèmes

Remarques

- Application au pied de la lettre de la récursivité
- On parle de méthode descendante (top-down) : le problème est résolu en résolvant des sous-problèmes, qui à leur tour sont résolus de la même façon et ainsi de suite...
- Il y a trois aspects fondamentaux : la Décomposition, les Appels Récursifs et la Fusion des solutions partielles
- Application récursive sur des tailles inférieures jusqu'à un certain seuil, cela garantit la finitude et la convergence

Divide-and-Conquer

Représentation algorithmique

```
1 Function divide_and_conquer(E,S)
2     if |E| ≤ n_0 then
3         simple(E, S)
4     else
5         (E_1, ..., E_k) ← divide(E)
6         for i ∈ [1..k] do
7             divide_and_conquer(E_i,S_i)
8         S ← merge(S_1, ..., S_k)
```

Exemple de Divide-and-Conquer

TRI

38	27	43	3	9	82	10

Exemple 3.1 (Recherche exhaustive naïve)

1. Générer tous les $n!$ permutations possibles
2. Valider chaque solution avec une fonction $\mathcal{O}(n)$

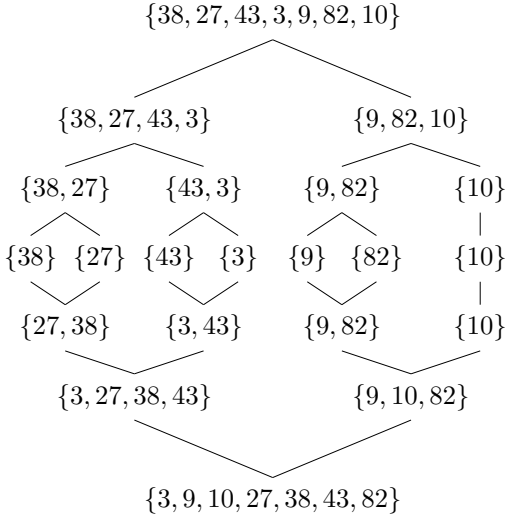
Exemple de Divide-and-Conquer

Tri fusion

```
1 Function merge_sort(E,g,d)
2     if d < g then
3         m ← (g + d)/2
4         merge_sort(E,g,m)
5         merge_sort(E,m + 1,d)
6         merge(E,g,m,d)
```

Exemple de Divide-and-Conquer

Exemple 3.2 (Tri fusion)



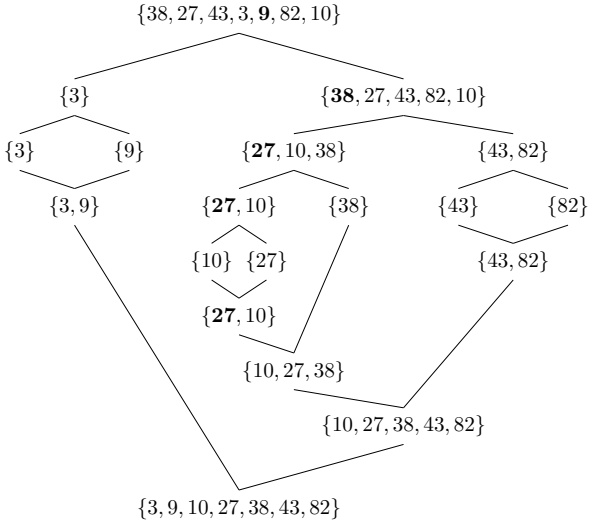
Exemple de Divide-and-Conquer

Tri rapide

```
1 Function quick_sort(E,g,d)
2     if d > g then
3         p ← pivot(E)
4         partition(E,g,d,p)
5         quick_sort(E,g,p − 1)
6         quick_sort(E,p + 1,d)
```

Exemple de Divide-and-Conquer

Exemple 3.3 (Tri rapide)



Complexité en temps

Soit $T(n)$ la complexité en temps pour une donnée de taille n, alors :

$$T(n) = \begin{cases} AS(n) & \text{si } n \leq n_0 \\ \sum_{i=1}^{k} T(n_i) + D(n) + F(n) & \text{sinon} \end{cases} \quad (4)$$

où

- $AS(n)$ est la complexité en temps de l'algorithme
- $D(n)$ est la complexité en temps de la décomposition
- $F(n)$ est la complexité en temps de la fusion
- $n_i = |E_i|$ est la taille du sous problème E_i, $i \in \{1, 2, \ldots, k\}$

Dans le cas général on ne peut rien dire de précis sur $T(n)$ mais il existe de nombreux résultats suivant les valeurs de $AS(n)$, $D(n)$ et $F(n)$ et la nature de la décomposition

Complexité en temps (suite.)

Théorème 3.4

L'équation de récurrence (où c est une constante positive)

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ T(n/2) + c & \text{si } n \geq 2 \end{cases} \tag{5}$$

admet comme solution $T(n) \in \Theta(\log_2 n)$

Complexité en temps (suite.)

Remarques

- Valide si $T(n) \leq \ldots$ (resp. $\geq$) avec $\mathcal{O}$ (resp. Ω) à la place de Θ
- Valide pour $n \leq n_0$ au lieu de $n = 1$ et $n > n_0$ à la place de $n \geq 2$
- Correspond à une stratégie récursive
  - résoudre un problème de taille n en ré-appliquant la même méthode sur le même problème de taille $n/2$ via un travail en $\mathcal{O}(1)$
- Si, dans un algo, on fait apparaître un paramètre dont dépend le nombre d'appels récursifs ou le nombre de boucles et dont la taille est divisée par 2 à chaque étape, il y aura $\Theta(\log_2 n)$

Complexité en temps (suite.)

Théorème 3.5

L'équation de récurrence (où a, b et c sont des constantes positives)

$$T(n) = \begin{cases} a & \text{si } n = 1 \\ c \cdot T(n/b) + a \cdot n & \text{si } n \geq 2 \end{cases} \tag{6}$$

admet comme solutions

$$T(n) \in \begin{cases} \Theta(n) & \text{si } b > c \\ \Theta(n \cdot \log n) & \text{si } b = c \\ \Theta(n^{\log_b c}) & \text{si } b < c \end{cases} \tag{7}$$

Complexité en temps (suite.)

Remarques

- Valide si $T(n) \leq \ldots$ (resp. $\geq$) avec $\mathcal{O}$ (resp. Ω) à la place de Θ
- Valide pour $n \leq n_0$ au lieu de $n = 1$ et $n > n_0$ à la place de $n \geq 2$
- Correspond à une stratégie récursive
  - résoudre un problème de taille n en le décomposant en c sous-problèmes de taille n/b puis résoudre ces c sous-problèmes de telle sorte que la décomposition ainsi que la construction de la solution globale se fasse en $\Theta(n)$
- À retenir absolument : le cas $T(n) = 2 \cdot T(n/2) + a \cdot n$ qui donne $T(n) \in \Theta(n \cdot \log n)$

Complexité en temps (suite.)

On peut généraliser le Théorème 3.5 :

Théorème 3.6

L'équation de récurrence (où a, b et c sont des constantes positives)

$$T(n) = \begin{cases} a & \text{si } n = 1 \\ c \cdot T(n/b) + a \cdot n^k & \text{si } n \geq 2 \end{cases} \tag{8}$$

admet comme solutions

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } b^k > c \\ \Theta(n^k \cdot \log n) & \text{si } b^k = c \\ \Theta(n^{\log_b c}) & \text{si } b^k < c \end{cases} \tag{9}$$

Complexité en temps (suite.)

Remarques

- Valide si $T(n) \leq \ldots$ (resp. $\geq$) avec $\mathcal{O}$ (resp. Ω) à la place de Θ
- Valide pour $n \leq n_0$ au lieu de $n = 1$ et $n > n_0$ à la place de $n \geq 2$
- Correspond à une stratégie récursive
  - résoudre un problème de taille n en le décomposant en c sous-problèmes de taille n/b puis résoudre ces c sous-problèmes de telle sorte que la décomposition ainsi que la construction de la solution globale se fasse en $\Theta(n^k)$

Complexité en temps (suite.)

Théorème 3.7

L'équation de récurrence (où a, α et β sont des constantes positives, et $\alpha + \beta \leq 1$)

$$T(n) = \begin{cases} a & \text{si } n = 1 \\ T(\alpha \cdot n) + T(\beta \cdot n) + a \cdot n & \text{si } n \geq 2 \end{cases} \tag{10}$$

admet comme solutions

$$T(n) \in \begin{cases} \Theta(n) & \text{si } \alpha + \beta < 1 \\ \Theta(n \cdot \log n) & \text{si } \alpha + \beta = 1 \end{cases} \tag{11}$$

Complexité en temps (suite.)

Remarques

- Valide si $T(n) \leq \ldots$ (resp. $\geq$) avec $\mathcal{O}$ (resp. Ω) à la place de Θ
- Valide pour $n \leq n_0$ au lieu de $n = 1$ et $n > n_0$ à la place de $n \geq 2$
- Correspond à une stratégie récursive
  - résoudre un problème de taille n en le décomposant en 2 sous-problèmes de taille $\alpha \cdot n$ et $\beta \cdot n$ puis résoudre ces 2 sous-problèmes de telle sorte que la décomposition ainsi que la construction de la solution globale se fasse en $\Theta(n)$

Complexité en temps (suite.)

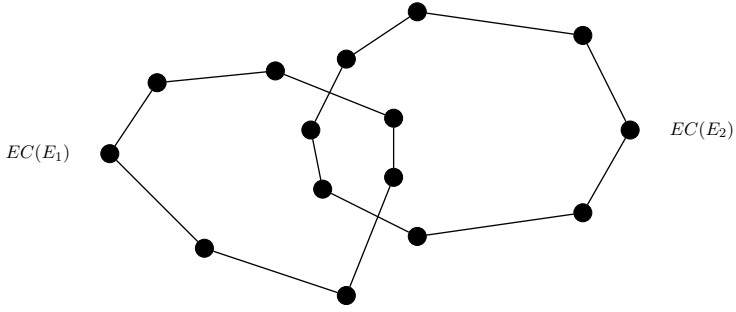
- On peut considérer ces résultats comme des pistes de recherche de stratégies de résolution de problèmes
  - e.g., si on cherche à résoudre un problème de taille n en $\mathcal{O}(n \cdot \log n)$ en temps, on peut essayer de le décomposer récursivement en 2 sous problèmes de taille $n/2$ de telle sorte que la décomposition et la construction de la solution globale se fasse en $\mathcal{O}(n)$ (cf. tri fusion)
- Pour l'implémentation il faudra déterminer expérimentalement un seuil n_0 en dessous duquel un algorithme plus simple est globalement plus efficace
- La complexité asymptotique dépend essentiellement des complexités de la décomposition et de la fusion
- Un bon équilibrage entre les sous-problèmes est souvent gage d'une bonne complexité (cf. tri fusion)
- Mais la meilleure complexité n'est pas forcément obtenue avec un parfait équilibrage, elle l'est quand la décomposition permet de réduire substantiellement la taille du problème (soit directement, soit par sous-problèmes) de n à $\alpha \cdot n$ (cf. Théorème 3.5 et Théorème 3.7), mais c'est difficile à obtenir !

Divide-and-Conquer pour EC

Algorithme de Shamos (1977)

- Décomposer $E = \{p_1, \ldots, p_n\}$ en $E_1 = \{p_1, \ldots, p_{n/2}\}$ et $E_2 = \{p_{n/2+1}, \ldots, p_n\}$
- Calculer récursivement $EC(E_1)$ et $EC(E_2)$
- Construire l'enveloppe convexe de E à partir de celles de E_1 et E_2 :

$$EC(E) = EC(EC(E_1) \cup EC(E_2)) \tag{12}$$

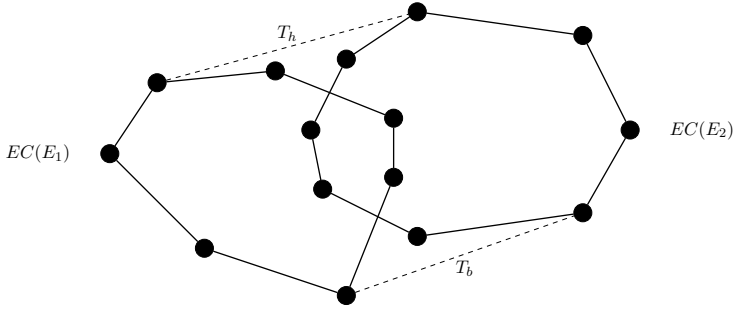


Divide-and-Conquer pour EC

Algorithme de Shamos (1977)

Résoudre l'équation (12) nécessite de

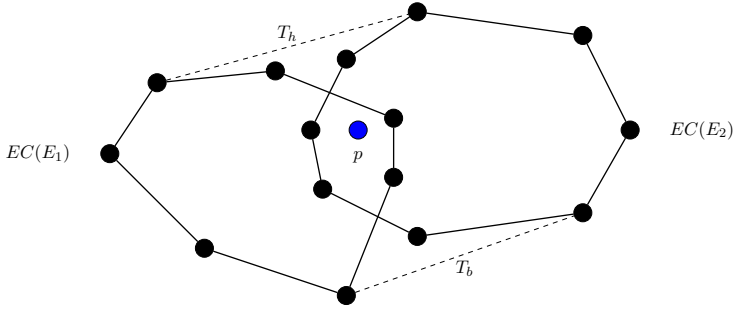
- Déterminer les deux segments T_b et T_h
- Supprimer des sommets « intérieurs »
- Raccorder aux extrémités de T_b et T_h



Divide-and-Conquer pour EC

Algorithme de Shamos (1977)

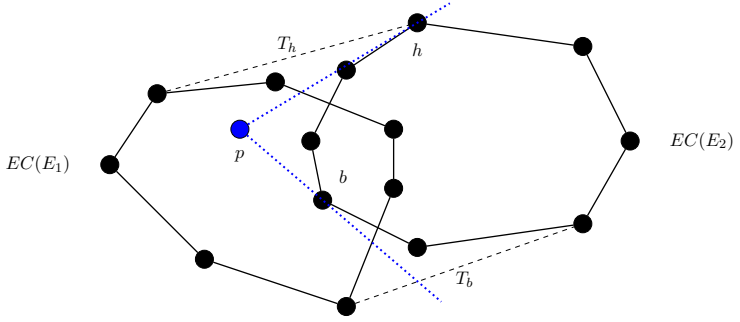
- Choix d'un point $p \in EC(E_1)$ (donc $p \in EC(E)$) en $\mathcal{O}(1)$
- Si $p \in EC(E_2)$ – vérifiable en $\mathcal{O}(|E_2|)$– alors, comme les sommets de E_1 et ceux de E_2 sont polairement ordonnés autour de p, il suffit de « fusionner » les deux listes
- Appliquer le balayage de Graham (cf. TD) sur la liste obtenue – $\mathcal{O}(|E_1| + |E_2|)$



Divide-and-Conquer pour EC

Algorithme de Shamos (1977)

- Si $p \notin EC(E_2)$ alors $EC(E_2)$ est contenue dans un secteur angulaire $\alpha < \pi$ d'origine p, s'appuyant sur les points b et h de $EC(E_2)$, en $\mathcal{O}(|E_2|)$
- Eliminer les points intérieurs à α sur $EC(E_1)$ et ceux situés entre h et b sur $EC(E_2)$
- Les points restants sont polairement ordonnés par rapport à p
- Fusionner ces points puis d'appliquer un balayage de Graham – en $\mathcal{O}(n)$



Divide-and-Conquer pour EC

Algorithme d'Eddy-Floyd

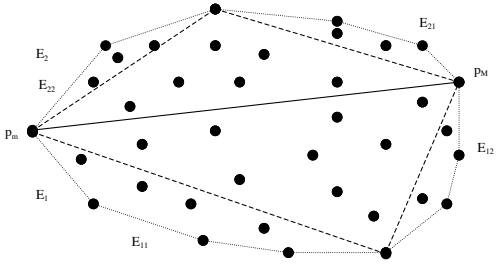
- Calcul des points extrémaux, p_m et p_M (d'abscisse minimale et maximale) de $E = \{p_1, \ldots, p_n\}$
- Détermination de E_1 et E_2, ensembles de points situés respectivement à droite et à gauche du segment $[p_m p_M]$
- Calcul de l'enveloppe inférieure (droite) sur E_1 de p_m à p_M (dans le sens direct) puis, par la même méthode, calcul de l'enveloppe supérieure (gauche) sur E_2 de p_M à p_m (dans le sens direct)
- L'enveloppe finale est obtenue par raccordement de ces deux sous-enveloppes

Divide-and-Conquer pour EC (suite.)

Algorithme d'Eddy-Floyd

Le calcul de l'enveloppe inférieure se fait en considérant E_1 :

- détermination d'un point p de E_1 tel que $p \in EC(E)$, par exemple le point situé à plus grande distance du segment $[p_m p_M]$ (ou le point p tel que l'aire du triangle $[p_m, p, p_M]$ soit maximale)
- Application récursive sur les sous-ensembles de points E_{11} et E_{12} de E_1 situés respectivement à droite des segments $[p_m p]$ et $[p p_M]$ (idem pour E_2)



Divide-and-Conquer pour EC (suite.)

Algorithme d'Eddy-Floyd

- L'étape préliminaire se fait en $\Theta(n)$ (calcul d'un minimum et d'un maximum)
- L'étape courante qui consiste à trouver p et à déterminer les deux sous-ensembles extérieurs au triangle se fait en $\mathcal{O}(n)$
- Chaque étape, sauf la première, trouve exactement un point de l'enveloppe convexe donc globalement l'algorithme est en $\mathcal{O}(n \cdot e)$ où $e = |EC(E)|$

Divide-and-Conquer pour EC

Remarques

- L'algorithme de Shamos s'apparente au Tri par Fusions alors que l'algorithme d'Eddy-Floyd est très proche du Tri Rapide
- Peut-on s'inspirer d'autres algorithmes de Tri pour trouver d'autres algorithmes de construction de l'EC dans le plan ? Et dans l'espace ?
- L'algorithme d'Eddy-Floyd est plus intéressant que celui de Shamos si l'on sait qu'il y aura peu de points sur $EC(E)$

Avantages et Inconvénients du Divide-and-Conquer

- Simplicité de l'approche et de nombreux problèmes se décomposent naturellement
- Preuve immédiate : finitude (car appels récursifs sur des problèmes de tailles inférieures) et validité (preuve par récurrence sur la taille) à condition que la décomposition et la fusion soient irréprochables
- Facilité de programmation dans un langage disposant de la récursivité
- Parallélisme possible quand les sous problèmes peuvent être résolus de façons indépendantes
- Un parfait équilibrage n'est pas toujours facile à obtenir, il peut être facilité par un prétraitement, un tri préalable des données, par exemple
- La récursivité peut amener des redondances dans le sens où les mêmes calculs peuvent être effectués plusieurs fois comme dans l'algorithme récursif résultant directement de l'équation de récurrence définissant la suite de FIBONACCI
- La détermination du seuil idéal n_0 dépend bien sûr du problème, de l'algorithme, de l'implémentation

Menu

Pré-traitement

On trouve aussi les appellations Pré-Conditionnement ou *Preprocessing*

Principe

Il s'agit d'organiser, d'ordonner, de structurer, . . . les données pour accélérer les traitements ultérieurs

Exemple 4.1

Imaginez un dictionnaire dont les mots sont listés dans un ordre quelconque. . .
La recherche d'un mot dans un dictionnaire est bien sûr efficace grâce au classement alphabétique, en gros en $\mathcal{O}(\log \log n)$ en moyenne, c'est en gros une Recherche par Interpolation.
Et le mot « *ordinateur* » ne signifie t'il pas aussi « qui ordonne », « qui met en ordre » ?

Pré-traitement (suite.)

Cette approche est capitale en informatique si l'on doit résoudre le même type de problèmes ou répondre à des questions identiques sur les mêmes données ou des données évoluant dans le temps.
On peut analyser algorithmiquement les stratégies de résolution de ce type de problèmes sous trois aspects :

- Temps de Prétraitement (Preprocessing Time), c'est-à-dire la complexité en temps de l'algorithme structurant les données
- Taille de la Structure de Données utilisée pour stocker les données
- Temps de Résolution (Query Time), c'est-à-dire temps de calcul nécessaire à la résolution du problème ou pour obtenir la réponse à une question

Examinons cette stratégie sur quelques problèmes

Pré-traitement sur un problème de recherche

Définition 4.2 (Recherche)

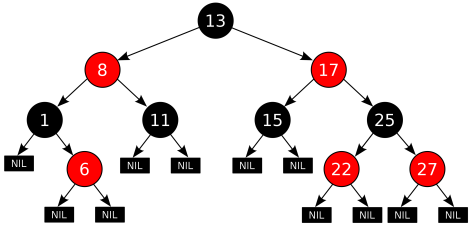
Données $E = \{e(1), e(2), \ldots, e(n)\}$, un ensemble de n éléments

Question Tester si un élément, x, de même nature que ceux de E, est dans E et, éventuellement, si oui, le supprimer x de E, et sinon le rajouter à E

Toute structure de données permettant de réaliser ces opérations s'appelle un Dictionnaire

- Un tri préalable, temps de prétraitement en $\Theta(n \log n)$ et structure de taille $\Theta(n)$, permet la Recherche Dichotomique (ou Binaire ou Logarithmique) en $\Theta(\log_2 n)$
- E est alors représenté par un tableau trié, ce qui pose des problèmes pour la suppression et l'insertion car ces opérations ne sont faisables qu'en $\mathcal{O}(n)$ dans le pire cas

Pré-traitement sur un problème de recherche (suite.)



Les meilleures structures pour ce problème sont les Arborescences (ou Arbres, en Informatique) de Recherche Equilibrées (AVL, Arbres 2-3, B-Arbres, Arbres Rouges-Noirs, ...) pour lesquelles on a (à une constante près) :

- un prétraitement en $n \cdot \log n$
- une structure de taille n
- la recherche (et la suppression, et l'insertion) en au plus $\log n$

On peut montrer que ces performances sont optimales, à une constante près

Pré-traitement pour P^4

Rappel : Soit $E = \{p_1, p_2, \ldots, p_n\}$, un ensemble de n points du plan, chacun étant donné par ses coordonnées $p_i = (x_i, y_i)$, il s'agit de trouver dans E deux points dont la distance est minimale

- La recherche exhaustive, qui teste toutes les paires de points, donne une solution en $\Theta(n^2)$ en temps et en $\Theta(n)$ en espace
- Sur des nombres, en dimension 1, on cherche deux nombres les plus proches : se résoud en $\mathcal{O}(n \log n)$
- On peut aussi résoudre P^4 dans le plan de façon optimale en $\mathcal{O}(n \log n)$
  - Pré-traitement : tri des points par rapport à x, puis par rapport à y
  - Divide-and-Conquer sur les points triés

Pré-traitement pour P^4 (suite.)

Principe du Divide-and-Conquer

- Division de E en deux sous ensembles E_1 et E_2 de même taille, $n/2$, par rapport à x
- Application récursive de l'algorithme sur E_1 et E_2
- Obtention de deux paires de points les plus proches, $\{A_1, B_1\}$ pour E_1 et $\{A_2, B_2\}$ pour E_2 et donc des deux distances correspondantes, $d_1 = d(A_1, B_1)$ et $d_2 = d(A_2, B_2)$

Comment alors déterminer une paire de points les plus proche, $\{A, B\}$ et leur distance, δ, dans E ?

- Remarque :
  - soit $\{A, B\} = \{A_1, B_1\}$
  - soit $\{A, B\} = \{A_2, B_2\}$
  - soit A est dans E_1 et B est dans E_2

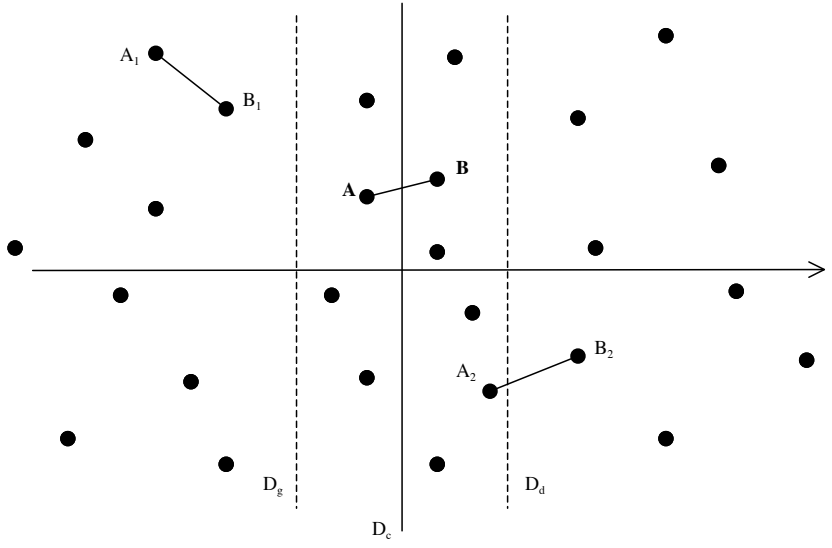
Pré-traitement pour P^4 (suite.) (suite.)

Principe du Divide-and-Conquer

- Notons $d = \min(d_1, d_2)$
- Où peuvent être A et B dans le dernier cas ?
- Soient p_i un point d'abscisse maximale de E_1 et p_j un point d'abscisse minimale de E_2
- Notons D_c la droite verticale d'équation $x = 1/2 \cdot (x_i + x_j)$ et D_g et D_d, les deux droites verticales d'équations respectives :
  - $x = 1/2 \cdot (x_i + x_j) - d$, à gauche
  - $x = 1/2 \cdot (x_i + x_j) + d$, à droite
- Il faut alors chercher A et B dans la bande verticale centrale de largeur $2d$ située à « cheval » sur E_1 et E_2 définie par D_g et D_d

Pré-traitement pour P^4 (suite.)

Géométriquement



Pré-traitement pour P^4 (suite.)

Problème : rechercher efficacement, en temps linéaire, le points A et B dans la bande verticale centrale

- Trier par rapport à l'ordonnée ne suffit pas car a priori chaque point doit être comparé au pire à $n-1$ autres points

Théorème 4.3

S'il y a $n \geq 5$ points dans un carré de coté égal à d alors au moins deux d'entre eux sont à une distance $< d$

Démonstration.

Découpons ce carré en quatre carrés, chacun ayant un coté égal à $d/2$, alors au moins 2 points parmi eux, p et q, se trouvent dans l'un d'entre eux, on a alors :

$$d(p,q) \leq \sqrt{\frac{d^2}{4} + \frac{d^2}{4}} = \sqrt{\frac{d^2}{2}} = \frac{d}{\sqrt{2}} < d \tag{12}$$

Ceci montre que pour tout point p de E_1 situé entre D_g et D_c il n'y aura qu'au plus quatre points de E_2 (situés entre D_c et D_d) à tester □

Pré-traitement pour P^4 (suite.)

Algorithme récursif

Principe

- Tri de E par x croissants puis par y croissants
- Construction de la liste E
- Parcours simultané des listes E_1 et E_2 et pour chaque point $p(x, y)$, vérifiant $x \geq x_g$ si $p \in E_1$ ou $x \leq x_d$ si $p \in E_2$, on calcule sa distance aux points de l'autre liste ayant une ordonnée plus grande et situés à une distance au plus δ, dès qu'une distance plus petite est trouvée on la stocke ainsi que les points correspondants
- On compare la distance ainsi trouvée à d_1 et d_2 et la réponse est la paire de points correspondant à la distance minimale, δ

Pré-traitement pour P^4 (suite.) (suite.)

Algorithme récursif

```
1  Function P^4(E,1,n ;A,B,δ)
2      if n ≤ n_0 then
3          simple(E,1,n ;A,B,δ)
4      else
5          m ← (1 + n)/2
6          E_1 ← ∅
7          E_2 ← ∅
8          for p_i ∈ E do
9              if k_i ≤ m then E_1 ← E_1 + {p_i} else E_2 ← E_2 + {p_i}
10             if k_i = m then i(g) ← k_i
11             if k_i = m + 1 then i(d) ← k_i
12         P^4(E_1, 1, m ; A_1, B_1, d_1)
13         P^4(E_2, m + 1, n ; A_2, B_2, d_2)
14         d ← min(d_1, d_2)
15         δ ← d
16         x_c ← 1/2 · (x_{i(g)} + x_{i(d)})
17         x_g ← x_c − d
18         x_d ← x_c + d
```

Pré-traitement pour P^4 (suite.)

Complexité

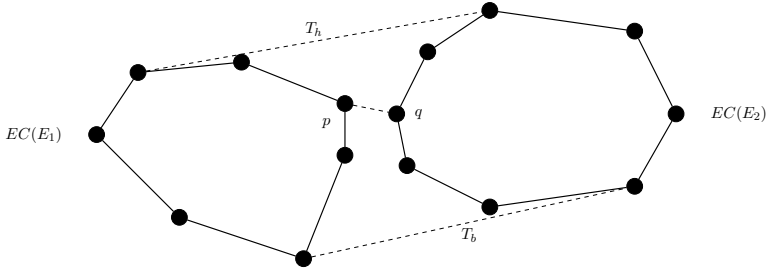
- La complexité en espace est en $\Theta(n)$ car l'algorithme ne manipule que des structures de tailles n
- Si l'on veut être rigoureux, il faut y rajouter la taille $\Theta(\log n)$, de la pile, invisible à nos yeux, gérant les appels récursifs
- La complexité en temps, $T(n)$, est en $\Theta(n \log n)$ car, en dehors du prétraitement qui est aussi en $\Theta(n \log n)$, $T(n)$ vérifie l'équation de récurrence :

$$T(n) = \begin{cases} c & \text{si } n \leq n_0 \\ 2 \cdot T(n/2) + c \cdot n & \text{si } n > n_0 \end{cases} \tag{13}$$

- La construction des listes E_1 et E_2 à partir de E se fait en $\Theta(n)$, il y a deux appels récursifs sur des données de tailles $n/2$, soit $2 \cdot T(n/2)$, puis la détermination de A, B et δ en $\Theta(n)$ et la suppression de E_1 et E_2 en $\Theta(n)$ car pour chaque point p retenu il y a au plus quatre points à tester

Pré-traitement pour EC

Le Tri préalable de points de $E = \{p_1, p_2, \ldots, p_n\}$ dans l'algorithme de Shamos permet de simplifier très agréablement l'opération de fusion et correspond à l'algorithme de Preparta-Hong (1979)

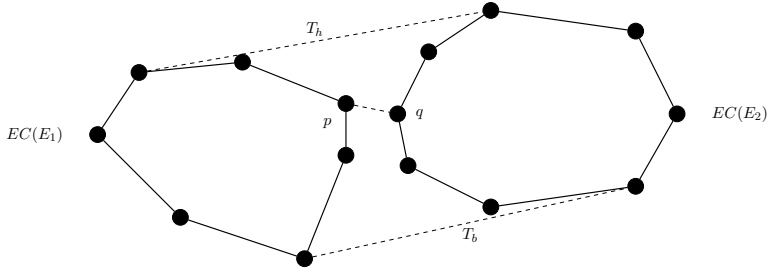


Pré-traitement pour EC (suite.)

- Soit p (ersp. q) le point d'abscisse extrémale de $EC(E_1)$ (resp. $EC(E_2)$)
- La boucle suivante permet facilement de trouver le segment T_h

```
1 while succ(p) ou pred(q) est à droite de [qp] do
2     if succ(p) est à droite de [qp] then
3         p ← succ(p)
4     q ← pred(q)
```

- $EC(E_1)$ et $EC(E_2)$ sont des listes doublement chaînées et ordonnées dans le sens direct, de leurs points extrémaux



Pré-traitement pour EC (suite.)

Complexité

- Le nombre d'étapes est en $\mathcal{O}(|EC(E_1)| + |EC(E_2)|)$, donc en $\mathcal{O}(n)$
- Il faut ensuite éliminer les points intérieurs situés entre T_b et T_h,
  - il suffit de parcourir $EC(E_1)$, puis $EC(E_2)$
  - et de « recoller les morceaux » pour obtenir $EC(E)$: $\mathcal{O}(|EC(E_1)| + |EC(E_2)|)$

Pré-traitement pour EC (suite.)

Algorithme

```
1 Function PH(E, 1, n)
2     if n ≤ n_0 then
3         return simple(E)
4     else
5         m ← (1 + n)/2
6         EC(E_1) ← PH({p_1, ..., p_m}, 1, m)
7         EC(E_2) ← PH({p_{m+1}, ..., p_n}, m + 1, n)
8         return EC(EC(E_1) ∪ EC(E_2))/* par la méthode précédente      */
```

Menu

Méthodes incrémentales

On parle aussi bien de Méthodes Incrémentales ou Itératives ou Séquentielles

Principe

Les éléments de la donnée, ou des données, sont traités les uns après les autres

Algorithme

```
1 Initialisation()
2 Prétraitement()/* cas offline                                    */
3 for i ∈ [1..n] do
4 |  Traitement(e(1), e(2), ..., e(i))
```

Complexité

Si n est la taille des données alors la complexité en temps est :

$$T(n) = \text{cout}(\text{Initialisation} + \text{Pretraitement} + \sum \text{Traitement}(e(1), ..., e(i))) \quad (14)$$

Méthodes incrémentales (suite.)

Remarques

- Méthode très simple et naïve, et toujours applicable
- Pas facile à mettre en œuvre efficacement et nécessitant, dans certains cas un prétraitement et des structures de données élaborées
- Certains problèmes ne peuvent se résoudre que de cette façon (e.g., Problèmes Online, temps réel)

Méthodes incrémentales (suite.)

Quelques exemples simples classiques

- Recherche Séquentielle d'un élément dans un ensemble, de complexités linéaires
- Sélection Séquentielle du minimum (ou du Maximum), la complexité en temps est linéaire et optimale
- Fusion séquentielle (ou simple) de deux suites triées (en listes ou en tableaux), là aussi la complexité en temps est linéaire et optimale
- Tri par Insertion : $e(1)$ étant trié, à la $i^{\text{ième}}$ étape $e(i)$ est inséré dans $e(1) \leq e(2) \leq \ldots \leq e(i-1)$ soit séquentiellement auquel cas l'algorithme est en $\mathcal{O}(n^2)$ ou alors en $\mathcal{O}(n \log n)$ si $\{e(1), \ldots, e(i)\}$ est structuré en Arbre de Recherche Equilibré
- Tri par Sélection Directe ou Arborescente (*Heap Sort*) (Cas offline)

Méthode incrémentale pour EC

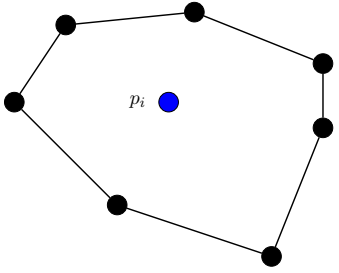
Si $E = \{p_1, p_2, \ldots, p_n\}$ est l'ensemble des points, l'application directe du principe donne l'algorithme suivant :

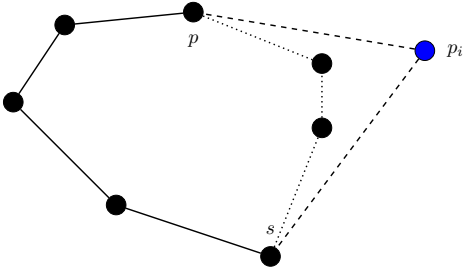
```
1 Function IterativeEC(E)
2     EC(E) ← {p_1}
3     for i ∈ [2..n] do
4         EC(E) ← EC(EC(E) ∪ {p_i})
5     return EC(E)
```

A chaque étape il s'agit de rajouter un point supplémentaire à l'enveloppe convexe courante

Méthode incrémentale pour EC (suite.)

- Notons C_i l'enveloppe convexe des i premiers points
- À la $i^{\text{ième}}$ étape : il faut rajouter p_{i+1} à C_i
- Il y a deux cas :
  1. soit p_{i+1} est intérieur à C_i : $C_{i+1} = C_i$
  2. soit p_{i+1} est extérieur à C_i : C_{i+1} est obtenu en trouvant les deux points d'appui p et s (tels que tous les points de C_i sont à gauche des segments $[pp_{i+1}]$ et $[p_{i+1}s]$) et en supprimant la partie de C_i située entre p et s, dans cet ordre





Méthode incrémentale pour EC (suite.)

- Ces opérations sont réalisables en $\mathcal{O}(|C_i|)$
- La complexité en temps, $T(n)$, vérifie donc :

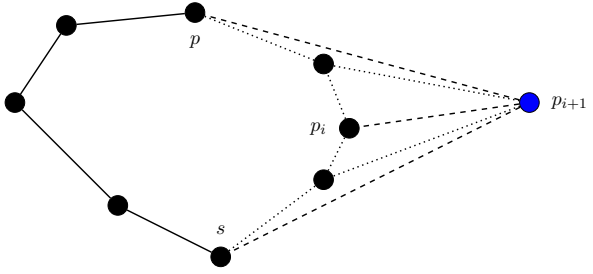
$$T(n) \leq \sum_{i=1}^{n} c \cdot |C_i| \leq \cdot \sum_{i=1}^{n} i \in \mathcal{O}(n^2) \tag{15}$$

Méthode incrémentale pour EC(suite.)

Cas Offline

- On peut améliorer la méthode en appliquant un prétraitement simple : on trie préalablement les points suivant leurs abscisses
- Notons $E = \{p_1, p_2, \ldots, p_n\}$ l'ensemble des points triés par abscisses croissantes, alors à la $i^{\text{ième}}$ étape on a :
- Les points étant triés,p_{i+1} est nécessairement à droite de C_i et p_i est le point le plus à droite de C_i

$\rightarrow$ Trouver les deux points p et s et de supprimer la partie de C_i allant de p à s



Méthode incrémentale pour EC(suite.)

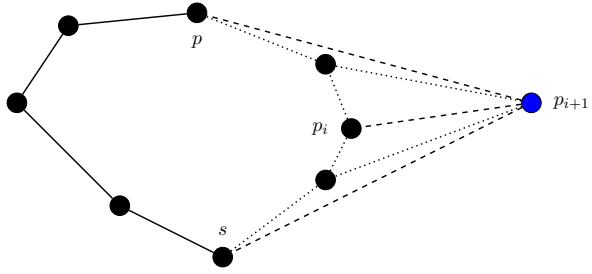
Cas Offline

- Détermination de s :

1 $s \leftarrow p_i$
2 **while** $succ(s)$ *est à droite du segment* $[p_{i+1}s]$ **do** $s \leftarrow succ(s)$;

- Détermination de p :

1 $p \leftarrow p_i$
2 **while** $pred(p)$ *est à droite du segment* $[pp_{i+1}]$ **do** $p \leftarrow pred(p)$;



Méthode incrémentale pour EC(suite.)

Cas Offline

Complexité en espace en $\Theta(n)$

- tableau ou liste des points en $\Theta(n)$
- + stockage de $EC(E)$

Complexité en temps en $\Theta(n \log n)$

- Le prétraitement se fait en $\Theta(n \log n)$
- Analysons l'étape de construction de $EC(E)$:
  - En première analyse on peut reprendre le raisonnement précédent, alors la complexité en temps, $C(n)$, de cette construction est :

$$C(n) \leq \sum_{i=1}^{n} c \cdot |C_i| \leq c \cdot \sum_{i=1}^{n} i \in \mathcal{O}(n^2) \tag{16}$$

  - En deuxième analyse on peut faire le raisonnement suivant : on va répartir les coûts sur chacun des n points
    - On compte « +1 » quand un point est testé et rejeté et l'on compte « +2 » pour chaque nouveau point (pour la détermination de p et s)
    - Ainsi tout point est considéré au plus trois fois, donc il y a au plus $3n$ étapes
    - La construction de $EC(E)$ est donc en $\Theta(n)$

Problème de l'EC – petit résumé

Pour le problème EC il existe au moins trois autres algorithmes :

- Graham (1972)
  - utilise un tri préalable (tri angulaire des points par rapport à un point de $EC(E)$)
  - et un calcul incrémental (Graham's scan, cf. TD)
  - $\mathcal{O}(n \cdot \log n)$
- Jarvis (1973)
  - algorithme incrémental
  - équivalent du Tri par Sélection
  - $\mathcal{O}(n \cdot e)$
- Kirkpatrick-Seidel (1986)
  - Divide and Conquer « très élaboré »
  - $\mathcal{O}(n \cdot \log e)$

Menu

Transformation et réduction

Principe

Il s'agit de reformuler le problème avant de le résoudre ou de le ramener à un problème déjà connu

Quelques exemples

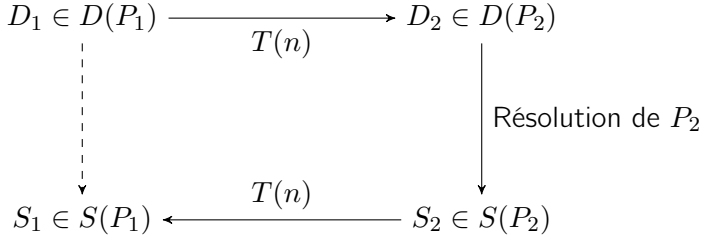
- Pour faire des opérations arithmétiques sur des chiffres romains, on les convertit en en décimal, on effectue les opérations en décimal puis on reconvertit les résultats en chiffres romains
- Toute opération réalisée sur ordinateur est en fait effectuée en binaire dans la machine puis est traduite en langage usuel compréhensible par les humains
- Toute multiplication usuelle se transforme en addition si l'on utilise les logarithmes (à condition d'avoir les tables de convertions)
- En Géométrie Algorithmique, de nombreuses transformations peuvent être utilisées, comme le passage des Coordonnées Cartésiennes aux Coordonnées Polaires ou la Dualité qui transforme les points en droites et réciproquement

Comparaisons (ou Réduction) de Problèmes

- Soient $\mathcal{M}$ un modèle de machine et μ une mesure de complexité et P_1, P_2 deux problèmes
- Notons n la taille de P_1 et $T(n)$ une fonction dépendant de n

Définition 6.1

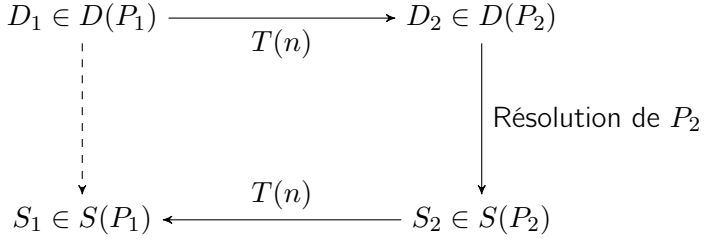
On dit que P_1 est $T(n)$-Transformable en P_2 ou $T(n)$-Réductible à P_2, noté $P_1 \leq_{T(n)} P_2$, si on a les transformations suivantes :



Comparaisons (ou Réduction) de Problèmes

Ce schéma signifie :

- Une donnée D_1 de P_1 est transformée en une donnée D_2 de P_2 en $\mathcal{O}(T(n))$
- la résolution de P_2 sur D_2 fournit la solution S_2 qui est transformée, en $\mathcal{O}(T(n))$, en solution S_1 de P_1, S_1 correspondant à la donnée D_1
- En gros on résout P_1 en utilisant (ou en passant par) P_2



Comparaisons (ou Réduction) de Problèmes(suite.)

Propriété 6.1

$$T_{P_2}(n) \in \mathcal{O}(f_2(n)) \Rightarrow T_{P_1}(n) \in \mathcal{O}(f_2(n) + T(n)) \tag{17}$$

Propriété 6.2

$$T_{P_1}(n) \in \Omega(f_1(n)) \Rightarrow T_{P_2}(n) \in \Omega(f_1(n) - T(n)) \tag{18}$$

Cette propriété va nous permettre d'obtenir des Bornes Inférieures sur des problèmes, en voici quelques exemples

Illustration sur un problème

Définition 6.3 (Problème d'unicité (PU))

Données $x_1, x_2, \dots x_n$, n nombres

Question Sont-ils tous distincts ?

Notons $\mathcal{M}$ une machine usuelle où, en $\mathcal{O}(1)$, on peut calculer une fonction polynomiale $f(x_1, x_2, \dots, x_n)$ de degré d sur les données

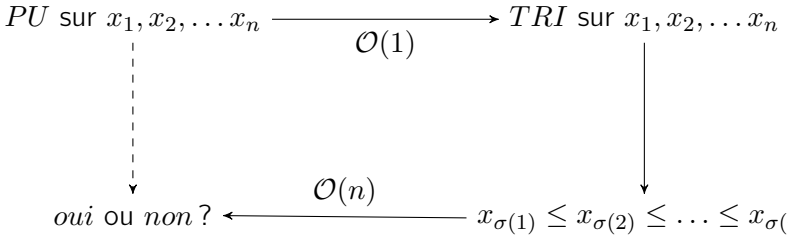
Théorème 6.4 (Ben-Or, 1983)

Dans le modèle $\mathcal{M}$, PU est en $\Omega(n \cdot \log n)$

Transformation de PU à TRI

Propriété 6.5

$$PU \leq_n TRI \tag{19}$$

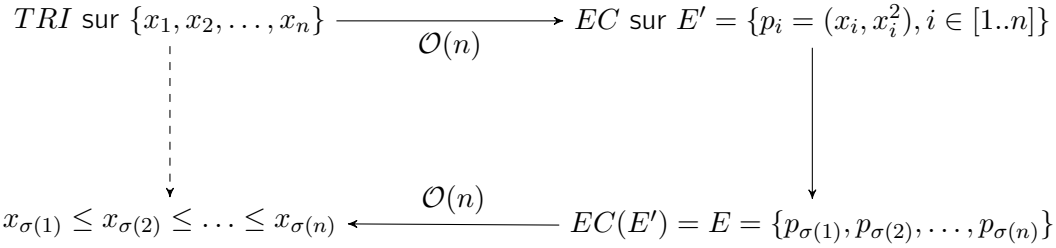


Pour répondre au PU il suffit de parcourir la liste des nombres triés en testant l'égalité de deux nombres consécutifs

Transformation de TRI à EC

Propriété 6.6

$$TRI \leq_n EC \tag{20}$$

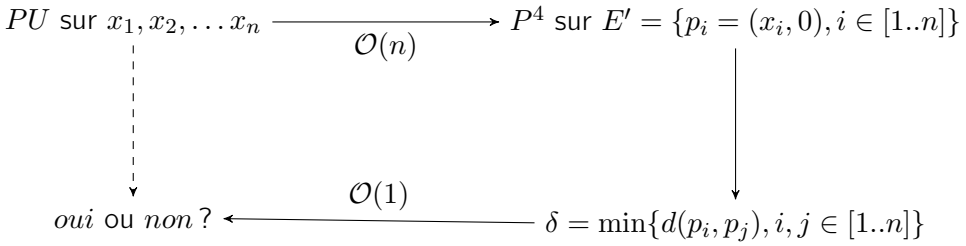


- Les n nombres sont transformés en un ensemble E' de n points se trouvant sur la parabole d'équation $y = x^2$
- $EC(E')$ est ordonnée dans le sens direct, de tous ces points car la courbe est convexe
- Pour obtenir les points triés il suffit de parcourir $EC(E')$ à partir du point d'abscisse minimale, cela se fait en $\mathcal{O}(n)$

Transformation de PU à P^4

Propriété 6.7

$$PU \leq_n P^4 \tag{21}$$



- Les nombres sont transformés en un ensemble E' de n points de l'axe des x
- La résolution de P^4 sur E' permet d'obtenir la distance minimale, δ, séparant deux points de E' et donc de répondre par oui ou non au problème de départ
- $\delta = 0$ correspond à la réponse non

Menu

Programmation dynamique

- Programmation dynamique = différentes techniques d'Optimisation Séquentielle
- Ne s'applique qu'à des problèmes qui peuvent se décomposer en sous-problèmes, dépendant les uns des autres
- Des sous-problèmes ont en commun des « sous-sous-problèmes »

Principe

Un algorithme de Programmation Dynamique résout chaque sous-sous-problème une seule fois et mémorise sa solution dans un tableau, une solution optimale du problème est obtenue à partir de solutions optimales de sous-problèmes

- C'est donc une méthode ascendante
- A opposer à la récursivité, méthode descendante (divide-and-conquer)

Programmation dynamique

Principe d'optimalité

- Il peut se formuler de la manière suivante :
  Dans une séquence optimale de décisions, quelle que soit la première décision, les décisions suivantes forment une sous-suite optimale, compte tenu des résultats de la première décision (principe de R. Bellman, initiateur de cette méthode vers 1950)
- On appelle souvent Politique la suite des décisions prises, le principe peut aussi s'énoncer : une politique optimale ne peut être formée que de sous-politiques optimales

Programmation dynamique

- Une des difficultés essentielles de la programmation dynamique est de reconnaître si un problème donné est justiciable du principe d'optimalité et peut être résolu par cette méthode
- Il est impossible de donner des recettes pour répondre à cette question
- On peut toutefois remarquer que le principe d'optimalité implique que le problème à étudier puisse être formulé comme celui de l'évolution d'un système

Méthodologie

1. Caractériser la structure d'une solution optimale
2. Définir par récurrence la valeur d'une solution optimale (mettre en évidence les liens entre les sous problèmes)
3. Calculer la valeur d'une solution optimale de manière séquentielle, ascendante (bottom-up)
4. Construire une solution optimale à partir des informations calculées

Programmation dynamique pour résoudre SAD

Pour rappel, SAD se traduit comme suit :

$$\max \sum_{i=1}^{n} x_i \cdot u_i$$
$$\text{t.q.} \sum_{i=1}^{n} x_i \cdot p_i \leq P$$
$$x_i \in \{0, 1\}$$

On suppose que $\forall i, p_i \leq P$ et $\sum_{i=1}^{n} p_i > P$

La recherche exhaustive, consistant à générer tous les sous-ensembles possibles, a une complexité en temps en $\mathcal{O}(n \cdot 2^n)$ dans le pire cas, et en $\mathcal{O}(2^n)$ en moyenne

Pour résoudre SAD par la Programmation Dynamique, il faut faire apparaître des sous-problèmes

Programmation dynamique pour résoudre SAD (suite.)

Soient les $n \cdot P$ sous-problèmes suivants, avec pour $i \in [1..n]$ et $j \in [1..P]$

Définition 7.1

On note $U(i, j)$ l'utilité maximale obtenue avec les i premiers objets et un poids j :

$$U(i,j) = \max \sum_{k=1}^{i} x_k \cdot u_k$$

$$\text{t.q.} \sum_{k=1}^{i} x_k \cdot p_k \leq j$$

$$x_k \in \{0, 1\}$$

La solution est bien sûr $U(n, P)$ ainsi qu'un contenu optimal donné par le vecteur booléen $(x_1, x_2, \ldots, x_n)$

- On suppose les objets numérotés de 1 à n, t.q. $p_1 \leq p_2 \leq \ldots \leq p_n$
- Par convention $U(i, k) = 0$ quand $0 \leq k \leq p_1 - 1$

Programmation dynamique pour résoudre SAD (suite.)

Propriété 7.2

$$U(1,j) = \begin{cases} 0 & \textit{si } j < p_1 \\ u_1 & \textit{sinon} \end{cases}$$

Propriété 7.3

Lorsque $2 \leq i \leq n$ *et* $p_1 \leq j \leq P$ *avec* $j - p_i \geq 0$ *on a :*

$$U(i,j) = \max\{U(i-1,j), U(i-1,j-p_i) + u_i\}$$

Programmation dynamique pour résoudre SAD (suite.)

L'idée de l'algorithme consiste donc à calculer :

1. d'abord les $U(1, j)$, quand $p_1 \leq j \leq P$
2. puis : $U(i, j) = \max\{(U(i-1, j), U(i-1, j-p_i) + u_i\}$, i variant de 2 à n, où $p_1 \leq j \leq P$ avec $j - p_i \geq 0$

Cela revient donc à remplir le tableau U, de taille $n \cdot (P - p_1 + 1)$, ligne après ligne et pour chaque colonne de p_1 à P

Programmation dynamique pour résoudre SAD (suite.)

Remplissage du tableau U

```
1  for j ∈ [p_1..P] do U(1, j) ← u_1
2  for i ∈ [2..n] do
3      for j ∈ [p_1..P] do
4          if j < p_i then
5              U(i, j) ← U(i − 1, j)
6          else
7              if j = p_i then
8                  U(i, j) = max{u_i, U(i − 1, j)}
9              else
10                 U(i, j) = max{U(i − 1, j), U(i − 1, j − p_i) + u_i}
```

La meilleure utilité est alors $U(n, P)$

Programmation dynamique pour résoudre SAD (suite.)

Comment récupérer une solution optimale ?

- Il faut pour cela stocker dans chaque case (i, j) (en plus de $U(i, j)$) la façon dont est obtenu le maximum, c'est-à-dire :
  - soit $x_i = 0$ (lorsque $U(i, j) = U(i - 1, j)$)
  - soit $x_i = 1$ (lorsque $U(i, j) = U(i - 1, j - p_i) + u_i$)
- Ainsi en « remontant » dans le tableau ligne après ligne à partir de la dernière case, $U(n, P)$, on a un vecteur optimal $(x_1, x_2, \ldots, x_n)$, ou politique optimale
- Cela nécessite n étapes supplémentaires

Programmation dynamique pour résoudre SAD (suite.)

Exemple 7.4

Voici un petit exemple où les utilités et les poids sont respectivement (10, 8, 5) et (6, 5, 4) avec un poids maximum de $P = 9$, il faut donc calculer :

$$\max_{4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 \leq 9} 5 \cdot x_1 + 8 \cdot x_2 + 10 \cdot x_3 \tag{22}$$

Programmation dynamique pour résoudre SAD (suite.)

Exemple 7.4

Voici un petit exemple où les utilités et les poids sont respectivement (10, 8, 5) et (6, 5, 4) avec un poids maximum de $P = 9$, il faut donc calculer :

$$\max_{4 \cdot x_1 + 5 \cdot x_2 + 6 \cdot x_3 \leq 9} 5 \cdot x_1 + 8 \cdot x_2 + 10 \cdot x_3 \tag{22}$$

On a un volume minimal de $p = 4$ (poids du plus petit objet), on a alors :

$U(i,j)$	4	5	6	7	8	9
1	5	5	5	5	5	5
2	5	8	8	8	8	13
3	5	8	10	10	10	13

- On obtient donc $U(3, 9) = 13$ avec le vecteur solution $(1, 1, 0)$
- L'utilité maximale est obtenue avec les deux premiers objets

Programmation dynamique pour résoudre SAD (suite.)

Estimations des complexités

- En temps
  - si on trie les poids il y a $\mathcal{O}(n \cdot \log n)$ étapes préliminaires
  - chaque ligne nécessite $P - p_1 + 1$ étapes (pour chaque case du tableau on calcule un maximum entre deux valeurs, le coût est donc constant) et il y a n lignes
  - Globalement, la complexité en temps est donc en $\mathcal{O}(n \cdot P)$, donc si P est « petit » l'algorithme peut être efficace
- En espace
  - le tableau a au plus $n \cdot P$ cases, chacune contenant deux valeurs ($U(i, j)$ et le x_i réalisant le maximum)
  - la complexité en espace est donc aussi en $\mathcal{O}(n \cdot P)$

Programmation dynamique s'applique dans de nombreux domaines

- La recherche des Plus Courts Chemins entre tous les couples de sommets d'un graphe valué
- La recherche d'une Triangulation Minimale d'un polygone convexe
- La construction d'Arbres Binaires Optimaux de Recherche
- La recherche de la Plus Longue Sous Séquence Commune à deux chaînes de caractères
- Le calcul de Plus Courtes Distances entre deux Mots
- La répartition d'une somme sur plusieurs projets d'Investissements pour obtenir la meilleure rentabilité.
- Un Programme de Production en vue de Minimiser les Coûts ou de Maximiser le Bénéfice
- ...

Et plus généralement la Programmation Dynamique permet de résoudre de nombreux problèmes de Répartitions, d'Investissements, de Productions, de Gestions de Stocks, d'Allocations de Ressources, etc.

Menu

Algorithmes gloutons

On les trouve aussi sous l'appellation d'algorithmes gourmands (greedy), ou voraces

Principe

Ce sont des algorithmes itératifs construisant une solution S à un problème P pas à pas : partant de $S = \emptyset$, on construit S en extrayant parmi les éléments non encore choisis, qu'on peut appeler candidats, le meilleur élément possible sans remettre en cause ce choix

Représentation algorithmique

1 $C \leftarrow$ {candidats possibles}
2 $S \leftarrow \emptyset$
3 **while** $C \neq \emptyset$ **do**
4 | Sélectionner le « meilleur » candidat possible $x = \arg\max_C f(S \cup \{x\})$
5 | $C \leftarrow C \setminus \{x\}$
6 | **if** $S \cup \{x\}$ *est une solution* **then** $S \leftarrow S \cup \{x\}$

Algorithmes gloutons (suite.)

Complexités

- La complexité en temps est a priori naturellement polynomiale car les éléments sont pris en compte les uns après les autres ou dans un certain ordre et elle dépend du
  - critère de sélection, celui optimisant $f(S \cup \{x\})$
  - coût du test « $S \cup \{x\}$ est-elle une solution ? »
- D'où des complexités de l'ordre de $\mathcal{O}(n)$ ou $\mathcal{O}(n \cdot \log n)$ ou $\mathcal{O}(n^2)$ voire $\mathcal{O}(n^3)$
- La complexité en espace est de l'ordre de $\mathcal{O}(n)$ ou de $\mathcal{O}(n^2)$

Algorithmes gloutons appliqués au PVC

Pour PVC, où il s'agit de trouver un cycle hamiltonien de longueur minimale passant par n points du plan

Exemple 8.1 (La plus petite arête)

On trie les arêtes suivant leur longueur (de la plus petite à la plus grande) et on les passe en revue dans cet ordre, une arête est gardée si elle ne forme pas de cycle (sauf la dernière) ni de sommet de degré trois

Exemple 8.2 (Le point le plus proche)

Partant d'un point quelconque, on rajoute à chaque étape le point le plus proche (choisi parmi les points non encore retenus) du dernier point choisi

Exemple 8.3 (La meilleure insertion)

Partant d'un cycle ayant trois sommets, à chaque étape on choisit parmi les sommets restants celui qui augmente le moins la longueur du cycle courant

Algorithmes gloutons appliqués au PVC (suite.)

Exemple 8.4 (La plus proche insertion)

Partant d'un cycle ayant trois sommets, à chaque étape, on choisit parmi les points restants le point le plus proche du cycle et on l'insère au mieux dans le cycle (entre deux sommets consécutifs tels que la longueur du cycle augmente le moins)

Exemple 8.5 (La plus lointaine insertion)

On part d'un cycle C ayant trois sommets, à chaque étape on choisit parmi les points restants celui dont la distance la plus proche à C est la plus grande et on l'insère au mieux dans C

De nombreux tests semblent indiquer que la dernière heuristique est la meilleure parmi les cinq proposées

Algorithmes gloutons appliqués au SAD

L'algorithme glouton classique pour SAD consiste à numéroter les objets dans l'ordre décroissant du rapport utilité sur poids et à les passer en revue dans cet ordre :

- si un objet rentre dans le sac on le garde
- s'il ne rentre pas on le rejette et on teste le suivant.

Cette heuristique est à la base de l'algorithme de Sahni (1975) et qui est un Schéma d'Approximation Polynomial (ensembles d'algorithmes permettant de s'approcher autant que l'on veut de l'optimal)

Algorithmes gloutons appliqués au SAD (suite.)

Pour SAD, si on suppose les objets triés par u_i/p_i décroissant, nous avons :

Théorème 8.6

Si on suppose uniquement $x_i \geq 0$ *alors la solution optimale est simplement* $x_1 = P/p_1$ *et* $x_i = 0$ *pour* $i \geq 2$

Algorithmes gloutons appliqués au SAD (suite.)

Théorème 8.7

Si on suppose $0 \leq x_i \leq 1$ *alors la solution optimale est :*

$$\begin{cases} x_i = 1 & \text{si } i \in [1..r] \\ x_{r+1} = \frac{1}{p_{r+1}} \cdot (P - \sum_{i=1}^{r} p_i) & \\ x_i = 0 & \text{si } i \in [r+2..n] \end{cases}$$

où $r = \min\{j, \sum_{i=1}^{j} p_i \leq P\}$

Remarquons qu'alors :

$$\sum_{i=1}^{n} x_i \cdot u_i = \sum_{i=1}^{r} u_i = \frac{u_{r+1}}{p_{r+1}} \cdot (P - \sum_{i=1}^{r} p_i)$$

$$\sum_{i=1}^{n} x_i \cdot p_i = \sum_{i=1}^{r} p_i = \frac{u_{r+1}}{p_{r+1}} \cdot (P - \sum_{i=1}^{r} p_i) = P$$

Algorithmes gloutons : remarques

- Les algorithmes gloutons sont en général simples à concevoir et à programmer
- Ces méthodes utilisent souvent un tri préalable (prétraitement) sur les objets manipulés
  - le tri est fait suivant un certain critère d'utilité
  - e.g. pour SAD : tri suivant le rapport utilité sur poids décroissant
- Les algorithmes gloutons sont à la base de nombreux algorithmes résolvant de façon approchée les problèmes NP-Difficiles
- Les algorithmes gloutons, bien que donnant toujours une solution réalisable, qu'on espère pas trop mauvaise, fournissent rarement une ou la solution optimale (c'est le cas pour les problèmes NP-Difficiles)
- Il y a cependant des problèmes polynomiaux où ils fonctionnent, bien que leur optimalité ne soit pas toujours facile à prouver...
- Il existe un théorie permettant d'expliquer pourquoi, dans certains cas, l'algorithme glouton donne toujours la solution optimale : la Théorie des Matroïdes (cf. support de cours)