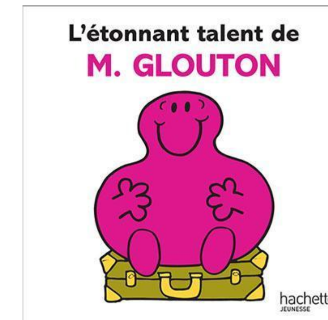
Algorithmes Gloutons

Sophie Tison
Master Informatique
Université de Lille

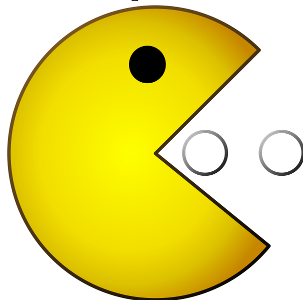
Trois paradigmes

- Diviser Pour Régner
- Programmation Dynamique
- Algorithmes gloutons (greedy algorithms)



Le principe d'une méthode gloutonne

Avaler tout ce qu'on peut
=
Construire au fur et à mesure une solution en faisant les choix qui paraissent optimaux localement



Dans certains cas, cela donnera finalement la meilleure solution: on parlera d'algorithmes gloutons exacts.

Dans d'autres, non, on parlera d'heuristiques gloutonnes.

Un premier exemple: Le monnayeur

On dispose des pièces de monnaie correspondant aux valeurs $\{a_0, \ldots, a_{n-1}\}$, avec $1 = a_0 < a_2 \ldots < a_{n-1}$.

Pour chaque valeur le nombre de pièces est non borné.

Etant donnée une quantité c entière, on veut trouver une façon de "rendre" la somme c avec un nombre de pièces minimum.

Un premier exemple: rendre la monnaie

Une instance du problème est la donnée des valeurs faciales des pièces $\{a_0, \ldots, a_{n-1}\}$, avec $1 = a_0 < a_2 \ldots < a_{n-1}$ et de la somme c à payer.

Une solution est pour chaque type de pièce ($0 \le i \le n-1$) un nombre de pièces nb_i telle que $\sum_{i=0}^{n-1} nb_i * a_i = c$: on a bien payé c (exactement);

Elle est optimale si $\sum_{i=0}^{n-1} nb_i$ est minimal: on a minimisé la fonction objectif, ici le nombre total de pièces.

Une instance du monnayeur

- Les pièces $a_0 = 1, a_1 = 2, a_2 = 5, a_3 = 10, a_4 = 20$
- la somme à payer $c = 26$

Solution optimale?

1 pièce de 20,

pas de pièce de 10,

1 pièce de 5,

pas de pièce de 2,

1 pièce de 1.

Peut-on faire mieux?

Pourquoi?

Comment justifier que la solution est optimale

Pour les pièces $a_1 = 1, a_2 = 2, a_3 = 5, a_4 = 10, a_5 = 20$ et la somme à payer $c = 26$, la solution: $20, 5, 1$ est-elle optimale? Preuve par cas?

- Si on utilise 1 pièce de 20, il reste 6 à payer: au moins deux pièces!
- Si on utilise 0 pièce de 20, il reste 26 à payer en 10, 5, 2, 1: au moins 3 pièces

Donc c'est optimal!

L'algorithme?

```
//c somme à payer entière >0
//valeurs des pièces: 1 = a[0] < ... < a[n − 1]
//les a[i] triés en croissant
int nb_pieces = 0;//solution vide
int[n] nb; //initialisé à 0
i = n − 1; nb + +;
while (c > 0) do
  | if ( c ≥ a[i]) then
  |   | nb[i] + +; nb_pieces + +; c = c − a[i];
  | else
  |   | i − −;
end
```

Autre version

```
//c somme à payer entière >0
//valeurs des pièces: 1 = a[0] < ... < a[n − 1]
//les a[i] triés en croissant
int nbtotal ← 0; //solution vide
int[n] nb;
for (int i = n − 1; i ≥ 0; i − −) do
    nb[i] ← c/a[i];
    nb_pieces ← nb_pieces + nb[i];
    c ← c  mod a[i];
end
```

L'algorithme est-il correct?

Par exemple, soit 26 à payer en pièces de 7, 6, 1

algo glouton: 3 Pièces de 7, 5 pièces de 1

solution optimale: 2 pièces de 7, 2 pièces de 6

Donc, Non, l'algo n'est pas correct!

L'algorithme est-il correct?

Un ancien système britannique: 1, 2, 6, 12, 24, 30, 60, 240

L'algorithme glouton était-il optimal?

Pour rendre 48, l'algorithme glouton n'est pas optimal.

On appelle canonique un système de pièces pour lequel l'algorithme glouton est optimal. Les systèmes actuels le sont (presque?) tous.

Quand est-il correct?

Par exemple, est-il toujours correct pour $1, 2, 5, 10, 20$?

La solution produite par l'algorithme glouton est

$g_{20} = c \div 20$

$g_{10} = (c \mod 20) \div 10$

$g_5 = ((c \mod 20) \mod 10) \div 5$ soit $g_5 = (c \mod 10) \div 5$

$g_2 = ((c \mod 10) \mod 5) \div 2$ soit $g_2 = (c \mod 5) \div 2$

$g1 = (c \mod 5) \mod 2$

Que dire de l'optimale?

Soit $o_{20}, o_{10}, o_5, o_2, o_1$ une solution optimale. On a $c = 20 * o_{20} + 10 * o_{10} + 5 * o_5 + 2 * o_2 + o_1$

Alors, $o_1 < 2$, sinon on peut remplacer 2 pièces de 1 par une de 2.

De même, $o_2 < 3$, sinon, on remplace 3 pièces de 2 par une de 5 et une de 1.

De plus si $o_2 = 2$, alors $o_1 = 0$, sinon on remplace deux de 2 et une de 1 par une de 5;

$o_5 < 2$, sinon, on remplace deux de 5 par une de 10.

$o_{10} < 2$, sinon, on remplace deux de 10 par une de 20.

Que dire de l'optimale?

$10 * o_{10} + 5 * o_5 + 2 * o_2 + o_1 <= 10 + 5 + 4 < 20$

$c = 20 * o_{20} + 10 * o_{10} + 5 * o_5 + 2 * o_2 + o_1$.

Donc $o_{20} = c \div 20 = g_{20}$ et $10 * o_{10} + 5 * o_5 + 2 * o_2 + o_1 = c \bmod 20$.

De même, $5 * o_5 + 2 * o_2 + o_1 < 10$ donc $o_{10} = c \bmod 20 \div 10 = g_{10}$.
Donc $5 * o_5 + 2 * o_2 + o_1 = c \bmod 10$;

Comme $2 * o_2 + o_1 < 5$, $o_5 = c \bmod 10 \div 5 = g_5$ et $2 * o_2 + o_1 = c \bmod 5$

De même, $o_2 = (c \bmod 5) \div 2 = g_2, o_1 = c \bmod 5 \bmod 2 = g_1$.

Le glouton est-il optimal?

Pour ce système $1, 2, 5, 10, 20$, la solution gloutonne est (l'unique) optimale.

Pour le système $1, 2, 5, 10, 20, 50$, la solution gloutonne est-elle optimale?

Le cas non canonique

Quand le système n'est pas canonique, comment calculer rapidement une façon optimale de rendre la monnaie?

Le principe général d'une méthode gloutonne

On est souvent dans le cadre de problèmes d'optimisation, plus précisément:

On a un ensemble fini E.

Une solution est construite à partir des éléments de E: c'est par exemple une partie de E ou un multi-ensemble d'éléments de E ou une suite (finie) d'éléments de E ou une permutation de E qui satisfait une certaine contrainte.

=la somme à payer

À chaque solution S est associée une fonction objectif v(S): on cherche une solution qui maximise (ou minimise) cette fonction objectif.

Le nombre de pièces

Le schéma général

Il est basé sur un critère local de sélection des éléments de E pour construire une solution optimale. En fait, on travaille sur l'objet " solution partielle"- "début de solution"- et on doit disposer de:

- select: qui choisit le meilleur élément restant selon le critère glouton.
- complete? qui teste si une solution partielle est une solution (complète).
- ajoutPossible? qui teste si un élément peut être ajouté à une solution partielle, i.e. si la solution partielle reste un début de solution possible après l'ajout de l'élément. Dans certains cas, c'est toujours vrai!
- ajout qui permet d'ajouter un élément à une solution si c'est possible.

Le schéma d'algorithme

```
// on initialise l'ensemble des "briques" des solutions.
Ens.init();
// on initialise la solution: ensemble (ou suite) "vide" ou..
Sol.Init();
while (Non Sol.complete?() et Ens.NonVide?()) do
    //on choisit x selon critère glouton
    x ← Ens.select();
    if Sol.ajoutPossible(x) then
        Sol.ajout(x); fsi;
    //dans certains problèmes, toujours le cas
    if CertainesConditions then
        Ens.retirer(x);
    // selon les cas, x considéré une fois ou plus
end
// la Solution partielle est a priori complète
return Sol;
```

Le schéma d'algorithme

```
// on initialise l'ensemble des "briques" des solutions.
Ens.init();
// on initialise la solution: ensemble (ou suite) "vide" ou..
Sol.Init();
while (Non Sol.complete?() et Ens.NonVide?()) do
    //on choisit x selon critère glouton
    x ← Ens.select();
    if Sol.ajoutPossible(x) then
        Sol.ajout(x); fsi;
    //dans certains problèmes, toujours le cas
    if CertainesConditions then
        Ens.retirer(x);
    // selon les cas, x considéré une fois ou plus
end
// la Solution partielle est a priori complète
return Sol;
```

Un exemple avec un raisonnement type

Le problème: sont données n demandes de réservation -pour une salle, un équipement...- avec pour chacune d'entre elles l'heure de début et de fin (on supposera qu'on n'a pas besoin de période de battement entre deux réservations).

Bien sûr, à un instant donné, l'équipement ne peut être réservé qu'une fois!

On cherche à donner satisfaction au maximum de demandes.

La formalisation du problème

Donnée:
n –le nombre de réservations
$(d_1, f_1), ..., (d_n, f_n)$ –pour chacune d'entre elles, le début et la fin-

Sortie: les réservations retenues i.e. $J \subset [1..n]$ tel que:

. elles sont compatibles :
$i \in J, j \in J, i \neq j \Rightarrow d_i >= f_j$ ou $f_i <= d_j$

. On satisfait le maximum de demandes, i.e. $|J|$ (le cardinal de J) est maximal.

Une instance

Donnée:
n, le nombre de réservations = 8
$(3, 8), (0, 6), (1, 4), (6, 8), (4, 7), (5, 11), (7, 9), (9, 10)$

Sortie:
~~(3, 8)~~, ~~(0, 6)~~, (1, 4), ~~(6, 8)~~, (4, 7), ~~(5, 11)~~, (7, 9), (9, 10)

Quel critère glouton?

- durée croissante?
- durée décroissante?
- début croissant?
- fin croissante?
- début décroissant?
- fin décroissante?

Quel critère glouton?

Symétrie

- durée croissante
- durée décroissante
- début croissant ou fin décroissante
- début décroissant ou fin croissante

Quel critère glouton?

Symétrie

- durée croissante
- durée décroissante
- début croissant ou fin décroissante
- début décroissant ou fin croissante

L'algorithme glouton

```
g=Vide;
Lib=0; //la ressource est dispo
nb=0; //nb de rés. acceptées
 trier les demandes par fin croissante;
 pour chaque demande i
  si d_i>=Lib
  //on peut la planifier
   {g.ajouter(i);
     Lib=f_i;
     nb++;}
```

coût= celui du tri = en $O(n \log n)$

Parenthèse

On aurait pu tester tous les sous-ensembles possibles de réservations.
Il y en a 2^n.

Comment prouver que l'algorithme est correct?

1. La solution est correcte, i.e. les réservations retenues sont bien compatibles: facile à vérifier par induction.
   On prend pour invariant { les demandes de g sont compatibles et finissent toutes au plus tard à Lib}.

2. La solution est optimale...?

Comment prouver que l'algo est correct?

On définit une notion de distance sur les solutions.

Une solution est une liste de demandes acceptées; on les suppose triées dans l'ordre des fins croissantes.

Soient deux solutions A, B différentes; soit i le plus petit indice tel que A_i soit différent de B_i ou tel que A_i soit défini et non B_i ou inversement. Alors on définit la distance de A et B par:

$$\mathrm{dist}(A, B) = 2^{-i}$$

Preuve de l'optimalité de l'algorithme

Raisonnons par l'absurde: Supposons que la solution gloutonne g ne soit pas optimale.

Soit alors o une solution optimale telle que $\mathrm{dist}(g, o)$ soit minimale (une telle o existe bien, l'ensemble des solutions étant fini).

Soit donc i telle que $\mathrm{dist}(g, o) = 2^{-i}$; il y a trois possibilités:

Cas 1

$g_i \neq o_i$

alors, la demande o_i a une date de fin au moins égale à celle de g_i: sinon comme elle est compatible avec les précédentes, l'algo glouton l'aurait examinée avant g_i et l'aurait choisie.

Mais alors, si on transforme o en o' en remplaçant l'activité o_i par g_i, on obtient bien une solution, de même cardinal que o donc optimale, et telle que $\mathrm{dist}(o', g) < \mathrm{dist}(o, g)$: on aboutit à une contradicition!

C'est ce qu'on appelle la propriété d'échange.

Cas 2 et 3

Cas 2: o_i n'est pas définie: alors $|o| < |g|$: contradiction, o ne serait pas optimale.

Cas 3: g_i n'est pas définie: impossible car l'activité o_i étant compatible avec les précédentes, l'algo glouton l'aurait choisie.

Donc, dans tous les cas, on aboutit à une contradiction; l'hypothèse "la solution gloutonne g n'est pas optimale" est fausse: la solution gloutonne est bien optimale!

Graphe d'intervalles

On peut modéliser les demandes par un graphe d'intervalles: un sommet est un intervalle, deux sommets sont reliés si les intervalles correspondants se chevauchent.

On cherche un ensemble indépendant maximal.

Aurait-on pu appliquer la programmation dynamique?

Oui, mais on aurait eu un algorithme un peu moins efficace et plus lourd.

Quelques variantes du problème

Que se passe-t-il si on veut maximiser la durée totale de réservation? Le critère glouton par durée décroissante marche-t-il?

Que se passe-t-il avec plusieurs ressources?

Complexité

La complexité est donc souvent de l'ordre de n logn (le tri selon le critère) $+n * f(n)$, si f(n) est le coût de la vérification de la contrainte et de l'ajout d'un élément. Les algorithmes gloutons sont donc en général efficaces...

Correction

Encore faut-il que l'algorithme donne bien une solution optimale... et qu'on sache le prouver...: là réside souvent la difficulté dans les algorithmes gloutons.

Les preuves de correction d'algorithmes gloutons, sont souvent basées sur une propriété appelée de type "échange":

Propriété d'échange: Soit une solution quelconque différente de la gloutonne: on peut la transformer en une autre solution au moins aussi bonne et "plus proche" de la solution gloutonne .

Glouton versus Dynamique

Dans les deux cas on peut se représenter l'ensemble des solutions sous forme d'un arbre, les solutions pouvant être vues comme une suite de choix.

Dans le cas de la programmation dynamique, on parcourt toutes les solutions mais on remarque que de nombreux noeuds de l'arbre correspondent aux mêmes sous-problèmes et l'arbre peut donc être élagué, ou plutôt représenté de façon beaucoup plus compacte comme un graphe (DAG: directed acyclic graph).

Dans le cas d'un algorithme glouton, on construit uniquement et directement-sans backtracking- une -et une seule- branche de l'arbre qui correspond à une solution optimale.

La Théorie

Les algorithmes gloutons sont basés sur la théorie des matroïdes.

Exemples classiques

- Les algorithmes de Prim et de Kruskal de calcul d'un arbre de recouvrement d'un graphe de poids minimal
- l'algorithme des plus courts chemins dans un graphe de Dijkstra
- le code de Huffman,
- de nombreuses versions de problèmes d'affectations de tâches...

Bilan

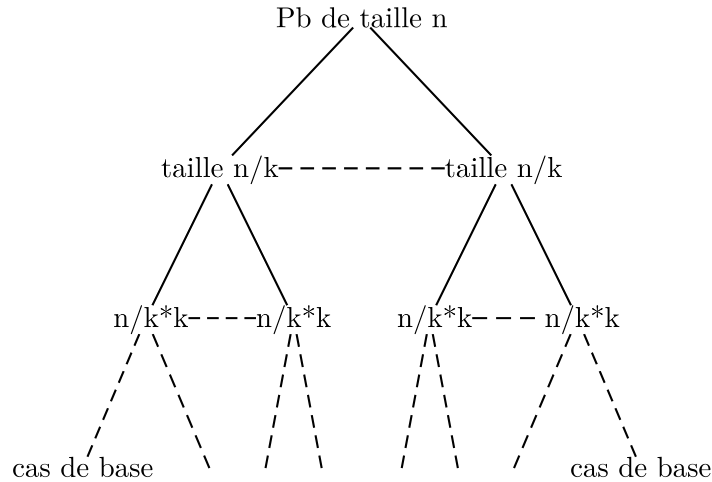
Pour mettre au point un algorithme glouton, il faut donc:

- Trouver un critère de sélection: souvent facile ... mais pas toujours
- Montrer que le critère est bon, c.à.d. que la solution obtenue est optimale: souvent dur!
- L'implémenter: en général facile et efficace!

Un problème, un critère

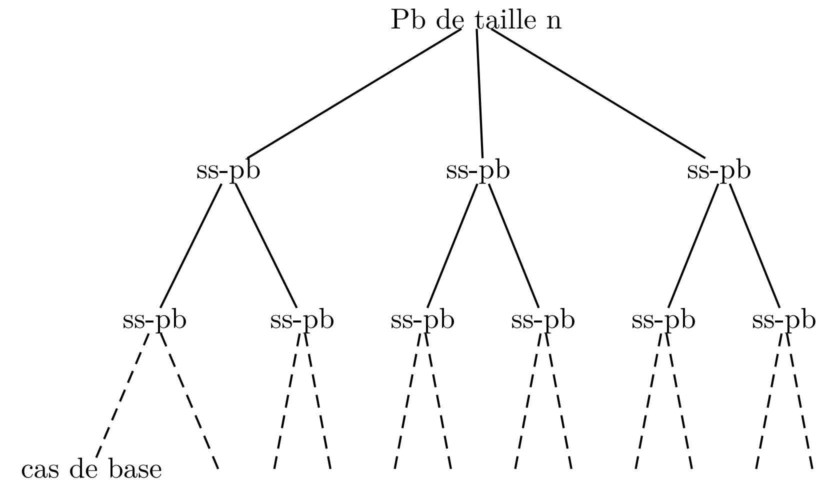
Bilan des 3 paradigmes

Diviser Pour Régner, Divide and Conquer



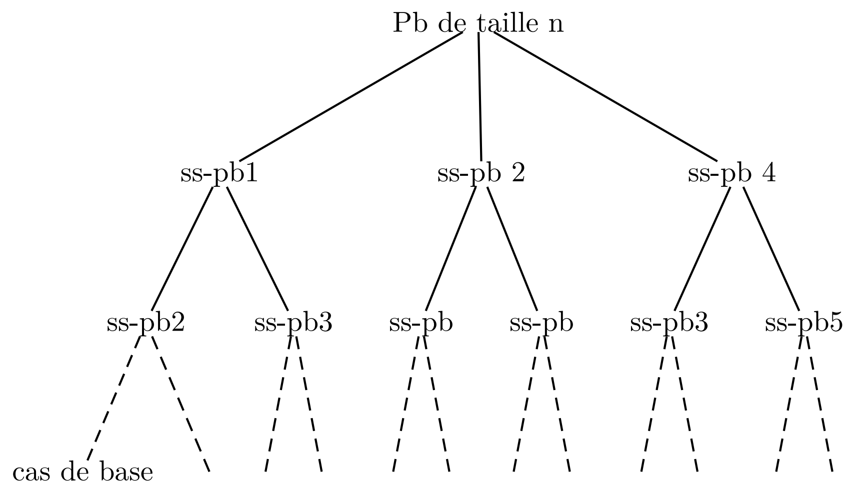
La hauteur de l'arbre est "petite", en $\log n$, la complexité dépend du nombre et de la taille des sous-problèmes à chaque étape, du coût de la recombinaison (Master Theroem)

Programmation Dynamique



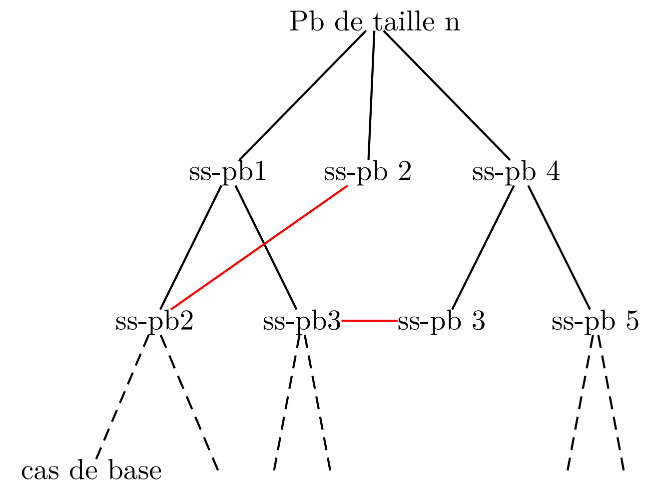
Il y a de nombreux sous-problèmes identiques.

Programmation Dynamique



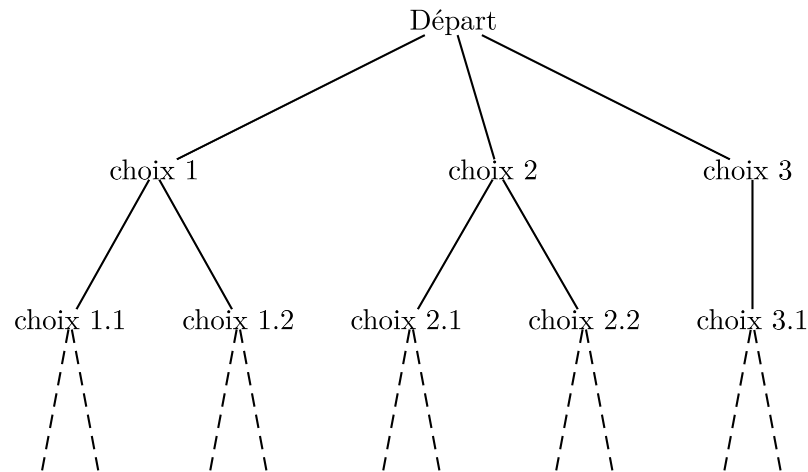
Il y a de nombreux sous-problèmes identiques.

Programmation Dynamique



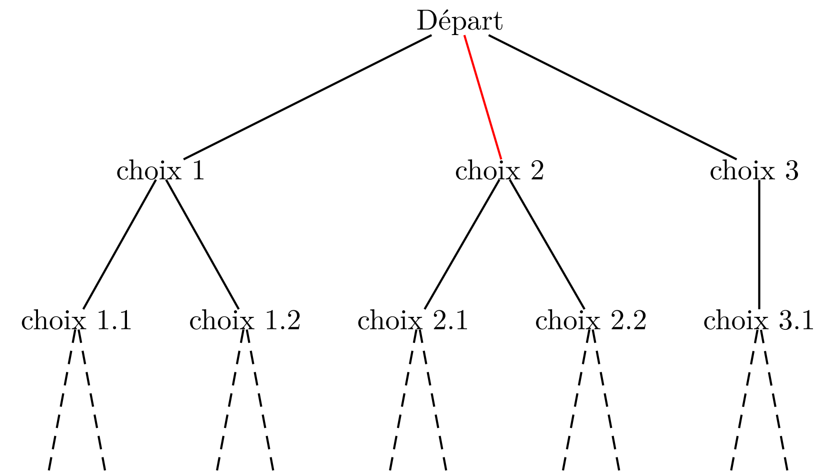
On transforme l'arbre en un graphe acyclique des sous-problèmes. Souvent de complexité polynomiale, mais parfois de degré assez élevé, parfois pseudo-polynomial.

Algorithmes Gloutons



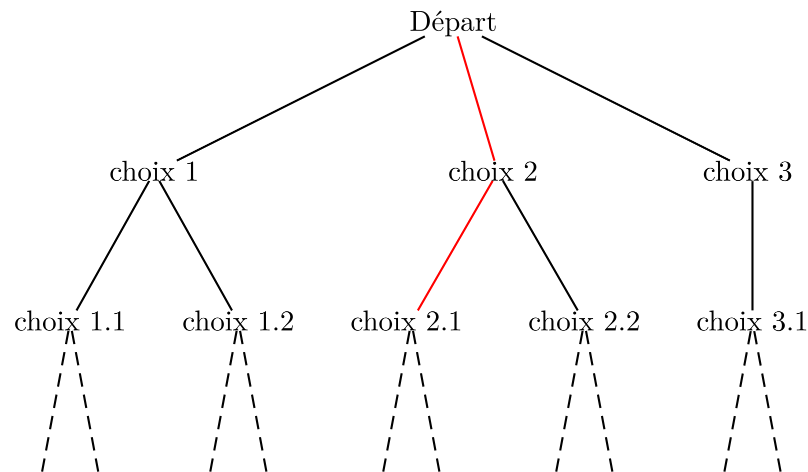
Le glouton fait ses choix localement pour ne pas explorer tout l'arbre.

Algorithmes Gloutons



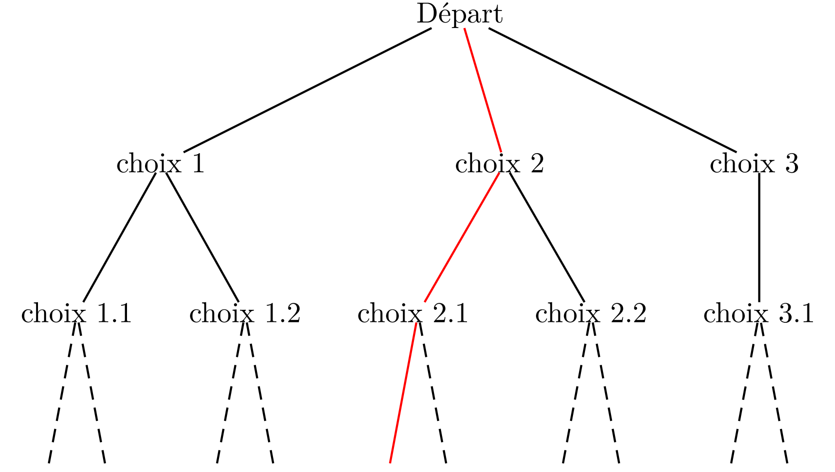
Le glouton fait ses choix localement pour ne pas explorer tout l'arbre.

Algorithmes Gloutons

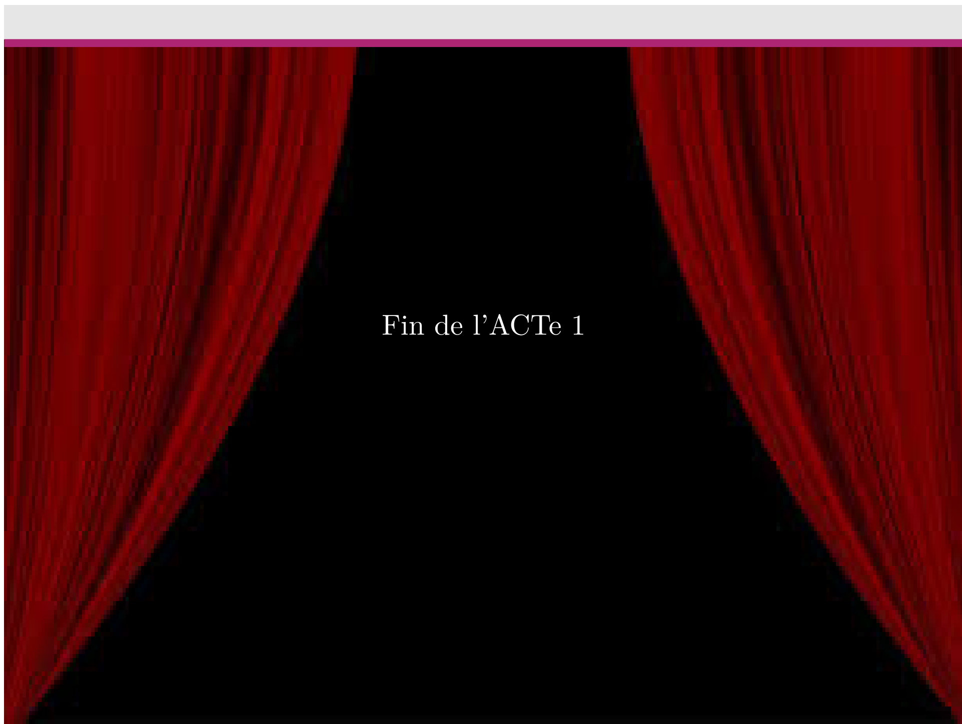


Le glouton fait ses choix localement pour ne pas explorer tout l'arbre.

Algorithmes Gloutons



Le glouton fait ses choix localement pour ne pas explorer tout l'arbre. Souvent très efficace et simple à écrire, mais la preuve nécessite un peu de travail.


Fin de l'ACTe 1