INF280
Stratégies de recherche

Antoine Amarilli

13 février 2017

Introduction

- **Stratégies classiques** pour résoudre les problèmes
- Se **demander** si l'une de ces stratégies peut marcher

$\rightarrow$ Réfléchir aussi au **nombre d'opérations** :

  - Environ 10^8–10^9 opérations par seconde
  - Exemple : Pour $n \simeq 10^4$ un algorithme en $O(n^2)$ passera…
  - … pour $n \simeq 10^6$ il ne passera pas, il faut $O(n)$ ou $O(n \log n)$

Table des matières

Principe

- **Énumérer** toutes les solutions possibles
- **Vérifier** en bloc si elles sont bonnes
- Attention à la **performance** !
  - **Nombre** de solutions
  - **Coût** de la vérification

Énumérer les permutations

```
int p[MAXN];
for (int i = 0; i < N; i++)
  p[i] = i;
do {
  // tester la permutation p
  // ...
} while (next_permutation(p, p+N));
```

Énumérer les sous-ensembles (N petit)

```
for (int s = 0; s < (1 << N); s++) {
  // s en binaire est un sous-ensemble de {0, ..., N-1}
  // (s & (1 << i)) pour savoir si i est dans l'ensemble
  // ...
}
```

Table des matières

Principe

- Ordre sur les choix
- Énumérer les options possibles
  - Pour chacune, la faire et tenter de continuer
- Si coincé, revenir en arrière (*backtrack*)

→ Permet de vérifier incrémentalement les solutions

→ Permet de rejeter les solutions partielles

Sudoku

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5		2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	4	3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	4	3	1		6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	4	3	1	??	6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	4	3	1		6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	4	3	??		6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	4	3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1		6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9		6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8		7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8	2	7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8	2	7	5
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8	2	7	5
9	7	1	3	8	5	6	2	4
5	4	3	7	2	6	8	1	9
6	8	2	1	4	9	7	5	3
7	9	4	6	3	2	5	8	1
2	6	5	8	1	4	9	3	7
3	1	8	9	5	7	4	6	2

Implémentation

- Souvent commode en récursif avec une structure globale
- Appel de fonction sur un choix : considérer chaque option :
  - Modifier la structure pour prendre l'option
  - Faire un appel récursif sur le choix suivant
  - En cas de réussite, réussir : laisse intacte la solution complète
  - Sinon, annuler la modification
- Attention à la taille de pile (quelques mégaoctets ou moins…)

  $\rightarrow$ Si la récursion est trop profonde, gérer manuellement la pile

Exemple : sudoku

```
int g[9][9];
int solve(int i) {
  int x = i/9, y = i%9;
  if (x >= 9) return 1;
  if (g[x][y] != 0)
    return solve(i+1);
  for (int k = 1; k <= 9; k++)
    if (acceptable(x, y, k)) {
      g[x][y] = k;
      if (solve(i+1))
        return 1;
      g[x][y] = 0;
    }
  return 0;
}
```

Cas particulier : minimax

- Retour sur trace usuel : si une branche réussit alors c'est réussi
- Cas du jeu à deux joueurs :
  - Une configuration est gagnée par le joueur au trait…
  - … si l'une des accessibles est gagnée par lui.
  - Inversement, si toutes les accessibles sont perdues pour lui…
  - … alors la configuration courante est perdante pour lui.

Exemple : minimax

```
int minimax(int i, int player) {
  int winner;
  if (winner = game_over())
    return winner;
  for (int m = 0; m < nmoves; m++)
    if (admissible(m, player)) {
      do_move(m, player);
      int ret = minimax(i+1, -player);
      undo_move(m, player);
      if (ret == player)
        return player;
    }
  return -player;
}
```

Heuristiques

- Ordre des **choix** :
  - Privilégier les choix **certains**
  - Privilégier les choix avec **peu d'options**
- Ordre des **options** :
  - Privilégier les options **contraignantes**

→ **Glouton** (cf plus tard) :
ordre pour lequel on ne revient jamais en arrière

Élagage (*Pruning*)

- **Booléen** : tout terminer dès qu'une solution est trouvée
- **Numérique** : couper les branches pires que l'optimum courant
- **Propagation de contraintes** :
  - observer les **conséquences** de l'option retenue
  - voir si on ne s'est pas **coincé** pour plus tard
- Pour minimax : **élagage alpha-beta**

Table des matières

Principe

- Rendre un algorithme **récursif** plus efficace

$\rightarrow$ Factoriser les **calculs déjà effectués**

Exemple : Fibonacci

```c
int fib(int i) {
  if (i == 0)
    return 0;
  if (i == 1)
    return 1;
  return fib(i-1) + fib(i-2);
}
```

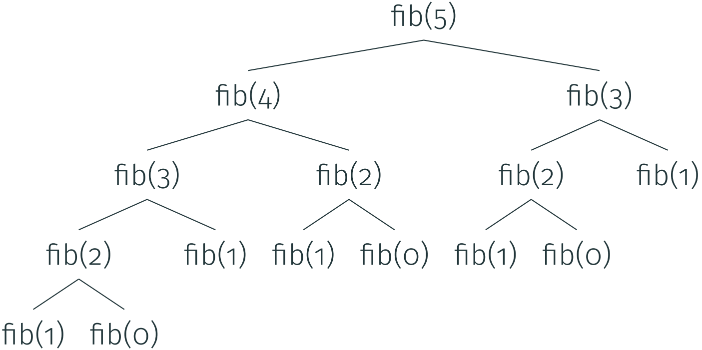
Exemple : Fibonacci

```c
int fib(int i) {
  if (i == 0)
    return 0;
  if (i == 1)
    return 1;
  return fib(i-1) + fib(i-2);
}
```

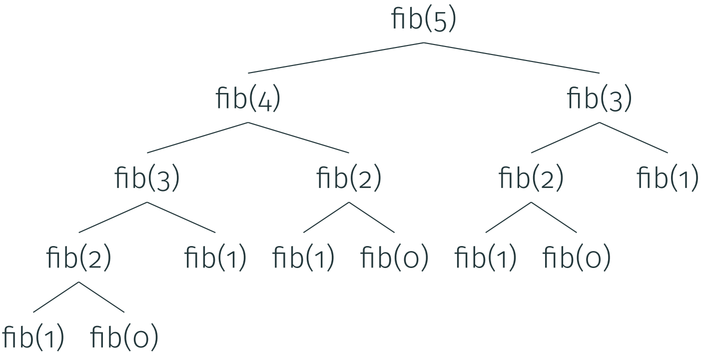
Mauvaises performances :

```
$ time ./a.out 42
./a.out 42  4.72s user 0.02s system 99% cpu 4.757 total
```

Inefficacité

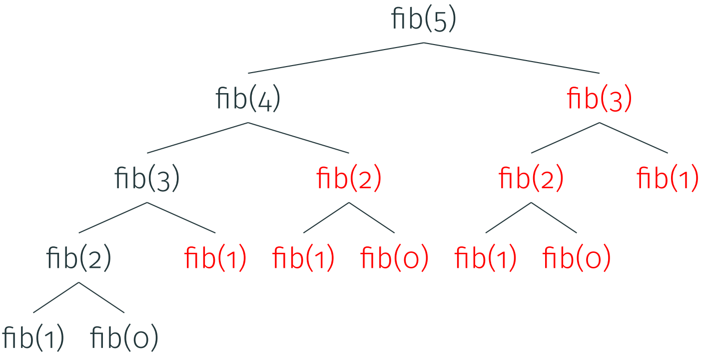


Inefficacité



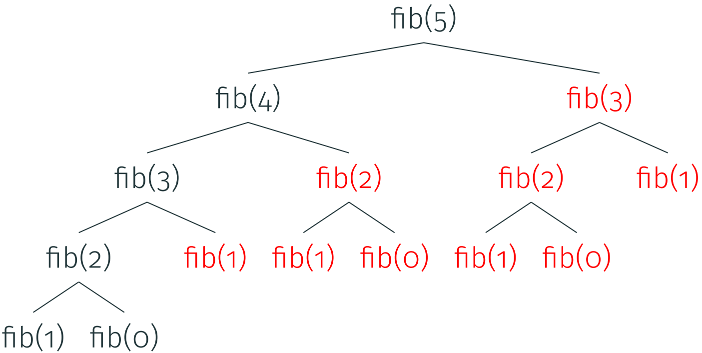
→ Exponentiel... peut-on faire mieux ?

Inefficacité



→ **Exponentiel...** peut-on faire mieux ?

Inefficacité



→ **Exponentiel...** peut-on faire mieux ?

→ Bien sûr, il y a de **meilleures solutions** pour Fibonacci

- Itératif (voir plus loin)
- Forme close, **exponentiation rapide**

Fibonacci en version mémoïsée

```
int M[100];

int fib(int i) {
  if (i == 0)
    return 0;
  if (i == 1)
    return 1;
  if (M[i])
    return M[i];
  return M[i] = fib(i-1) + fib(i-2);
}
```

Fibonacci en version mémoïsée

```c
int M[100];

int fib(int i) {
  if (i == 0)
    return 0;
  if (i == 1)
    return 1;
  if (M[i])
    return M[i];
  return M[i] = fib(i-1) + fib(i-2);
}
```

Mieux!

```
$ time ./a.out 42
./a.out 42  0.00s user 0.00s system 0% cpu 0.002 total
```

Pièges

- Choisir la bonne **valeur non initialisée** (ici, 0)
- Penser à **remettre à cette valeur** entre les inputs s'il le faut
- Ne pas oublier les **cas de base**
- Ne pas oublier d'**écrire dans le tableau**
- Choisir les bonnes **dimensions maximales**
- Si les valeurs ne sont **pas denses**, utiliser un (unordered)_map
- Attention à la **pile** ! (récursion non terminale)
- Il ne faut pas d'**état** ! (peut-on mémoïser le sudoku ?)

Autre exemple : distance d'édition de Levenshtein

- **Distance** pour passer d'une chaîne à l'autre
- **Opérations :**
  - **Insertion :** chat → chant
  - **Délétion :** cochon → cocon
  - **Substitution :** poule → poupe

→ Combien d'opérations successives nécessaires au **minimum** ?

Idée pour Levenshtein

- Pour passer d'une chaîne non vide s à une chaîne non vide t…
  - Soit les deux derniers caractères sont les mêmes : retirer
  - Soit le dernier caractère de t a été inséré
  - Soit le dernier caractère de t est le résultat d'une substitution sur le dernier caractère de s
  - Soit le dernier caractère de t est un caractère précédent de s et le dernier caractère de s a été supprimé

Idée pour Levenshtein

- Pour passer d'une chaîne non vide s à une chaîne non vide t...
  - Soit les deux derniers caractères sont les mêmes : **retirer**
  - Soit le dernier caractère de t a été **inséré**
  - Soit le dernier caractère de t est le résultat d'une **substitution** sur le dernier caractère de s
  - Soit le dernier caractère de t est un caractère précédent de s et le dernier caractère de s a été **supprimé**

- $D(i, j)$: distance du **préfixe** $s[0 \cdots i]$ au **préfixe** $t[0 \cdots j]$
- $D(i, j)$ est le **minimum** de :
  - $1 + D(i, j - 1)$: **insertion**
  - $1 + D(i - 1, j)$: **délétion**
  - $D(i - 1, j - 1)$ + 1 si $s[i] \neq s[j]$ et 0 sinon : **substitution**
- **Cas de base :** $D(0, j) = j$ et $D(i, 0) = i$

Exemple pour Levenshtein

		n	i	c	h	e	s
c							
h							
i							
e							
n							

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 $\bullet$	1 $\leftarrow$	2 $\leftarrow$	3 $\leftarrow$	4 $\leftarrow$	5 $\leftarrow$	6 $\leftarrow$
c							
h							
i							
e							
n							

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 •	1 ←	2 ←	3 ←	4 ←	5 ←	6 ←
c	**1** ↑						
h							
i							
e							
n							

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 ●	1 ←	2 ←	3 ←	4 ←	5 ←	6 ←
c	1 ↑	1 ↖					
h							
i							
e							
n							

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 ●	1 ←	2 ←	3 ←	4 ←	5 ←	6 ←
c	1 ↑	1 ↖	2 ↖				
h							
i							
e							
n							

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 •	1 ←	2 ←	3 ←	4 ←	5 ←	6 ←
c	1 ↑	1 ↖	2 ↖	2 ↖			
h							
i							
e							
n							

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 •	1 ←	2 ←	3 ←	4 ←	5 ←	6 ←
c	1 ↑	1 ↖	2 ↖	2 ↖	3 ←		
h							
i							
e							
n							

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 •	1 ←	2 ←	3 ←	4 ←	5 ←	6 ←
c	1 ↑	1 ↖	2 ↖	2 ↖	3 ←	4 ←	
h							
i							
e							
n							

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 ●	1 ←	2 ←	3 ←	4 ←	5 ←	6 ←
c	1 ↑	1 ↖	2 ↖	2 ↖	3 ←	4 ←	5 ←
h							
i							
e							
n							

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 •	1 ←	2 ←	3 ←	4 ←	5 ←	6 ←
c	1 ↑	1 ↖	2 ↖	2 ↖	3 ←	4 ←	5 ←
h	2 ↑	2 ↖	2 ↖	3 ↖	2 ↖	3 ←	4 ←
i	3 ↑	3 ↖	2 ↖	3 ↖	3 ↑	3 ↖	4 ↖
e	4 ↑	4 ↖	3 ↑	3 ↖	4 ↖	3 ↖	4 ↖
n	5 ↑	4 ↑	4 ↑	4 ↑	4 ↖	4 ↑	4 ↖

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 •	**1 ←**	**2 ←**	3 ←	4 ←	5 ←	6 ←
c	1 ↑	1 ↖	2 ↖	**2 ↖**	3 ←	4 ←	5 ←
h	2 ↑	2 ↖	2 ↖	3 ↖	**2 ↖**	3 ←	4 ←
i	3 ↑	3 ↖	2 ↖	3 ↖	**3 ↑**	3 ↖	4 ↖
e	4 ↑	4 ↖	3 ↑	3 ↖	4 ↖	**3 ↖**	4 ↖
n	5 ↑	4 ↑	4 ↑	4 ↑	4 ↖	4 ↑	**4 ↖**

Exemple pour Levenshtein

		n	i	c	h	e	s
	0 •	**1 ←**	**2 ←**	3 ←	4 ←	5 ←	6 ←
c	1 ↑	1 ↖	2 ↖	**2 ↖**	3 ←	4 ←	5 ←
h	2 ↑	2 ↖	2 ↖	3 ↖	**2 ↖**	3 ←	4 ←
i	3 ↑	3 ↖	2 ↖	3 ↖	**3 ↑**	3 ↖	4 ↖
e	4 ↑	4 ↖	3 ↑	3 ↖	4 ↖	**3 ↖**	4 ↖
n	5 ↑	4 ↑	4 ↑	4 ↑	4 ↖	4 ↑	**4 ↖**

n	**i**	**c**	**h**		**e**	**s**
		|	|		|	|
		c	**h**	**i**	**e**	**n**

Implémentation de Levenshtein

```
int M[100][100];
char *s, *t;

int dist(int i, int j) {
  if (!i)
    return j;
  if (!j)
    return i;
  if (M[i][j] >= 0)
    return M[i][j];
  return M[i][j] = min(
      dist(i-1, j-1) + ((s[i-1] == t[j-1]) ? 0 : 1),
      min(1 + dist(i, j-1),
          1 + dist(i-1, j)));
}
```

Table des matières

Principe

- **Mémoïsation :** traiter chaque sous-problème utile **une seule fois** à partir des plus gros
- **Dynamique :** traiter **tous** les sous-problèmes (même inutiles) des plus petits aux plus grands
- **Avantages des dynamiques :**
  - Souvent plus **efficaces** (pas de récursion), pas de **taille de pile**
  - Plus facile de réduire la **mémoire** en ne gardant pas tout
  - Parfois plus naturel de **voir le calcul**
- **Inconvénients :**
  - Calcule même les sous-problèmes **inutiles**
  - Plus difficiles à adapter à partir d'un **retour sur trace**

Exemple : Fibonacci

```c
int T[100];

int fib(int n) {
  T[0] = 0;
  T[1] = 1;
  for (int i = 2; i <= n; i++)
    T[i] = T[i-1] + T[i-2];
  return T[n];
}
```

Exemple : Fibonacci

```
int T[100];

int fib(int n) {
  T[0] = 0;
  T[1] = 1;
  for (int i = 2; i <= n; i++)
    T[i] = T[i-1] + T[i-2];
  return T[n];
}
```

- Bien sûr, c'est idiot !
- Inutile de tout garder en mémoire

Exemple : Fibonacci (amélioré)

```
int T[3];

int fib(int n) {
  T[0] = 0;
  T[1] = 1;
  for (int i = 2; i <= n; i++)
    T[i%3] = T[(i-1)%3] + T[(i-2)%3];
  return T[n%3];
}
```

Exemple : Fibonacci (amélioré)

```
int T[3];

int fib(int n) {
  T[0] = 0;
  T[1] = 1;
  for (int i = 2; i <= n; i++)
    T[i%3] = T[(i-1)%3] + T[(i-2)%3];
  return T[n%3];
}
```

- Mieux qu'un récursif mémoïsé !

Exemple : Levenshtein

```c
int M[100][100];
char *s, *t;

int dist(int ni, int nj) {
  for (int j = 0; j <= nj; j++)
    M[0][j] = j;
  for (int i = 1; i <= ni; i++) {
    M[i][0] = i;
    for (int j = 1; j <= nj; j++)
      M[i][j] = min(min(1 + M[i][j-1], 1 + M[i-1][j]),
            M[i-1][j-1] + ((s[i-1] == t[j-1]) ? 0 : 1));
  }
  return M[ni][nj];
}
```

Exemple : Levenshtein

```c
int M[100][100];
char *s, *t;

int dist(int ni, int nj) {
  for (int j = 0; j <= nj; j++)
    M[0][j] = j;
  for (int i = 1; i <= ni; i++) {
    M[i][0] = i;
    for (int j = 1; j <= nj; j++)
      M[i][j] = min(min(1 + M[i][j-1], 1 + M[i-1][j]),
          M[i-1][j-1] + ((s[i-1] == t[j-1]) ? 0 : 1));
  }
  return M[ni][nj];
}
```

- On peut à nouveau faire mieux

Exemple : Levenshtein (amélioré)

```c
int M[100][100];
char *s, *t;

int dist(int ni, int nj) {
  for (int j = 0; j <= nj; j++)
    M[0][j] = j;
  for (int i = 1; i <= ni; i++) {
    M[i%2][0] = i;
    for (int j = 1; j <= nj; j++)
      M[i%2][j] = min(min(1 + M[i%2][j-1], 1 + M[(i-1)%2][j]),
          M[(i-1)%2][j-1] + ((s[i-1] == t[j-1]) ? 0 : 1));
  }
  return M[ni%2][nj];
}
```

Table des matières

Principe

- Algorithme **glouton** : faire le choix localement meilleur
- Ne jamais **revenir** sur ses choix

Exemple de glouton : bicolorier un graphe

Entrée
- graphe orienté G
- racine *root*

Hypothèse Tous les sommets de G sont accessibles depuis *root*

Sortie Déterminer si G est biparti ?

Exemple de glouton : bicolorier un graphe

Entrée
- graphe orienté G
- racine *root*

Hypothèse Tous les sommets de G sont accessibles depuis *root*

Sortie Déterminer si G est biparti ?

$\rightarrow$ **Intuition :** si G est biparti, alors bipartition unique

- … excepté la symétrie entre les parties 1 et 2

Exemple de glouton : code

```
int color(int v, int c) {
  if (col[v])
    return col[v] == c ? 1 : 0;
  col[v] = c;
  for (unsigned int i = 0; i < adj[v].size(); i++)
    if (!color(adj[v][i], -c))
      return 0;
  return 1;
}

for (int i = 0; i < N; i++)
  col[i] = 0;
color(root, 1);
```

Gloutons et tris

- Importance de l'ordre des choix
- Parfois, glouton seulement possible pour le bon ordre
- Souvent, il faut trier pour avoir le bon ordre
- Se demander : le glouton suivant tel tri est-il optimal ?

Exemple de glouton et tri : choix d'activités

Entrée Activités avec date de début et de fin $d_i < f_i$

Sortie Sous-ensemble maximal sans chevauchement

Exemple de glouton et tri : choix d'activités

Entrée Activités avec date de début et de fin $d_i < f_i$

Sortie Sous-ensemble maximal sans chevauchement

$\rightarrow$ Intuition : trier les activités par date de fin croissante

Exemple de glouton et tri : code

```
for (int i = 0; i < N; i++) {
  scanf("%d %d", &d, &f);
  v.push_back(make_pair(f, d));
}
sort(v.begin(), v.end());
int nok = 0, last = -1;
for (int i = 0; i < N; i++) {
  if (v[i].second < last)
    continue;
  nok++;
  last = v[i].first;
}
```

Exemple de glouton et tri : justification

- Considérons une solution **optimale**
- Considérons le **tri** par date de fin croissante
- Considérons la première activité a que l'optimale **ne prend pas**
- On peut remplacer l'**activité suivante** de l'optimale par a
- On obtient une solution
  - **aussi bonne** que l'optimale
  - qui **fait le choix glouton**
- **Induction**, on répète le processus

$\rightarrow$ La solution gloutonne est une solution **optimale**

Exemple de glouton et tri : justification

- Considérons une solution **optimale**
- Considérons le **tri** par date de fin croissante
- Considérons la première activité a que l'optimale **ne prend pas**
- On peut remplacer l'**activité suivante** de l'optimale par a
- On obtient une solution
  - **aussi bonne** que l'optimale
  - qui **fait le choix glouton**
- **Induction**, on répète le processus

$\rightarrow$ La solution gloutonne est une solution **optimale**

Généralisations à des **activités pondérées** :

`https://en.wikipedia.org/wiki/Activity_selection_problem`

Table des matières

Principe

- Dichotomie :
  - $\rightarrow$ recherche d'un élément dans un tableau trié
- Généralisation :
  - $\rightarrow$ recherche d'une frontière entre deux régions
- Coût seulement logarithmique (contre-intuitif !)

Exemple de dichotomie : couverture de points

Entrée.
- points P sur un segment
- nombre K de points

Sortie. K points P' telle que la distance maximale d'un point de P à un point de P' soit minimale.

Exemple de dichotomie : couverture de points

Entrée.
- points P sur un segment
- nombre K de points

Sortie. K points P' telle que la distance maximale d'un point de P à un point de P' soit minimale.

$\rightarrow$ Idée : dichotomiser sur la distance maximale

Exemple de dichotomie : suite

- Une distance maximale est soit réalisable soit irréalisable
- Monotonie :
  - si d est réalisable alors tout $d' < d$ l'est aussi
  - si d est irréalisable alors tout $d' > d$ l'est aussi

  $\rightarrow$ une seule frontière entre réalisable et irréalisable
- Pour une distance maximale D fixée, algorithme glouton

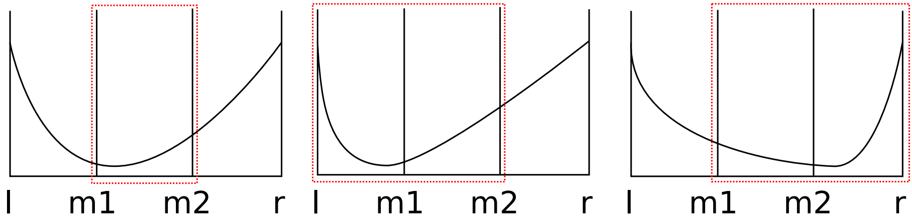
$\rightarrow$ Dichotomie puis glouton

Trichotomie (ternary search)

- Dichotomie : recherche d'une valeur cible
- Que faire si la cible est un optimum inconnu ?
- Hypothèse : la fonction a un seul optimum local

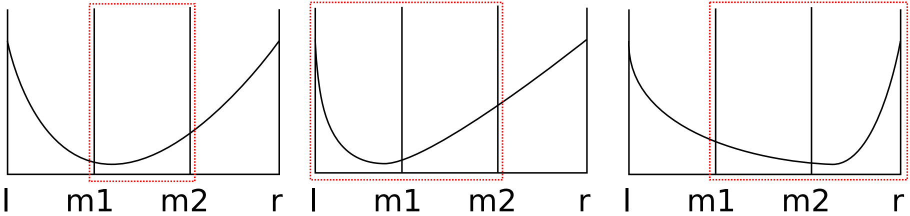
Trichotomie (ternary search)

- **Dichotomie** : recherche d'une **valeur cible**
- Que faire si la cible est un **optimum inconnu** ?
- **Hypothèse** : la fonction a **un seul optimum local**



Trichotomie (ternary search)

- **Dichotomie** : recherche d'une **valeur cible**
- Que faire si la cible est un **optimum inconnu** ?
- **Hypothèse** : la fonction a **un seul optimum local**



→ **Recherche ternaire**

Table des matières

Conclusion

- Gardez à l'esprit ces schémas classiques
- Bien sûr, combinaisons :
  - → Force brute sur un choix puis glouton sur les autres
  - → Dichotomie sur un paramètre puis glouton avec le bon tri
  - → etc.

Crédits

- Transparent 9 : `https://tex.stackexchange.com/a/43234`