La supériorité de l'ordre supérieur

Jean-Christophe Filliâtre
LRI, Université Paris Sud

Journées Francophones des Langages Applicatifs
28 janvier 2002

Introduction

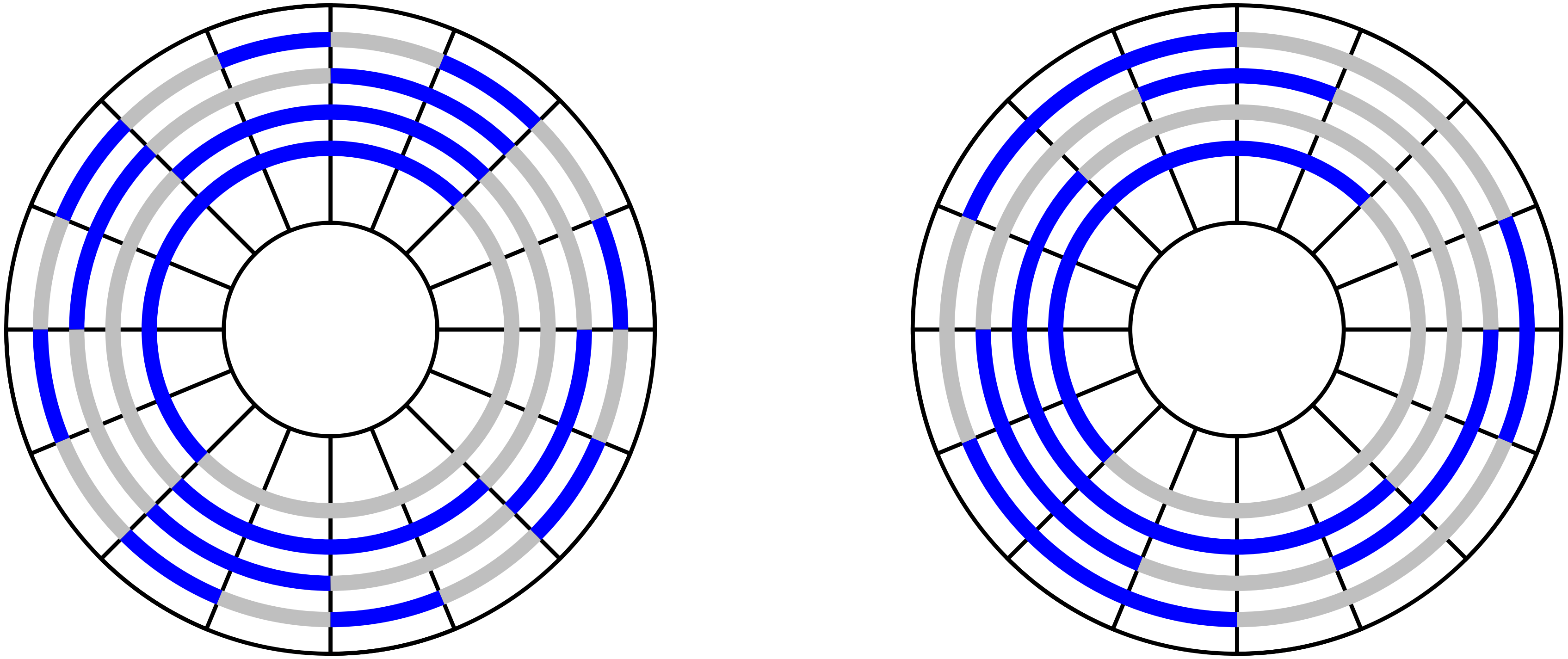
The Art of Computer Programming, Donald E. Knuth

Volume 4 : Combinatorial Algorithms, en préparation

- Volume 4A, Enumeration and Backtracking
  - 7.2. Generating all possibilities
    - 7.2.1.1 Generating all n-tuples ← sur le web

Codes de Gray

0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, . . .

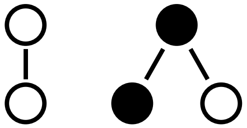
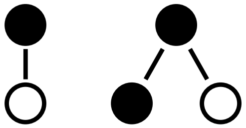


Généralisation à des forêts

Y. Koda et F. Ruskey

A Gray Code for the Ideals of a Forest Poset

(*J. Algorithms* **15**, 1993)

…,  ,  , …

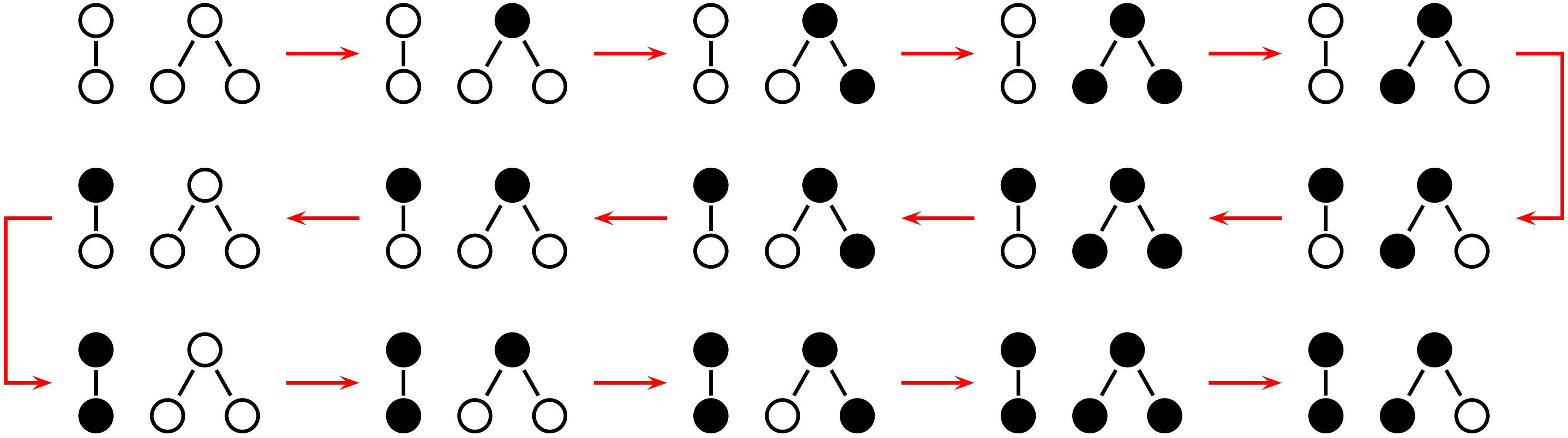
Plan

1. L'algorithme de Koda-Ruskey
2. De l'ordre supérieur
3. Encore plus d'ordre supérieur
4. Encore moins d'ordre supérieur
5. Travail en cours et perspectives

I

L'algorithme de Koda-Ruskey

Illustration



Implantations

Koda et Ruskey : deux pseudo-codes Pascal, dont un de complexité optimale

Knuth : deux implantations en CWEB

- *coroutines* : branchements `goto` (50 lignes)
- *loopless* : structures chaînées complexes (80 lignes)

II

De l’ordre supérieur

Arbres, forêts et coloriages

```
type tree   = Node of int × forest
and forest = tree list
```

```
1        4
|       / \
0      2   3
```

```
[ Node (1, [Node (0, [])]);
  Node (4, [Node (2, []); Node (3, [])]) ]
```

```
type color = Black | White
val bits : color array
```

Un code par continuations

On passe une fonction qui sera appliquée pour chaque coloriage

```
val loop_f : (unit → unit) → forest → unit
```

```
val loop_t : (unit → unit) → tree → unit
```

Coloriage des forêts

```
let rec loop_f k = function
  | [] → k ()
  | t :: f → loop_t (fun () → loop_f k f) t
and ...
```

Coloriage des arbres

```
...
and loop_t k (Node (i,f)) =
  if bits.(i) = White then begin
    k (); bits.(i) ← Black; loop_f k f
  end else begin
    loop_f k f; bits.(i) ← White; k ()
  end
```

Un premier code C_1

```
let rec loop_f k = function
  | [] → k ()
  | t :: f → loop_t (fun () → loop_f k f) t
and loop_t k (Node (i,f)) =
  if bits.(i) = White then begin
    k (); bits.(i) ← Black; loop_f k f
  end else begin
    loop_f k f; bits.(i) ← White; k ()
  end
```

Application à une forêt `f`

```
loop_f (fun () → (* affichage *)) f
```

Complexité

n = nombre de nœuds

N = nombre de coloriages

C_1 a une complexité $O(N)$ en espace

$O(nN)$ en temps

III

Encore plus d'ordre supérieur

Encore plus d'ordre supérieur

On retourne maintenant la fonction de coloriage

```
val loop_f : (unit → unit) → forest → (unit → unit)

val loop_t : (unit → unit) → tree → (unit → unit)
```

Un second code C_2

```
let rec loop_f k = function
  | [] → k
  | t :: f → loop_t (loop_f k f) t
and loop_t k (Node (i,f)) =
  let lf = loop_f k f in
  fun () →
    if bits.(i) = White then begin
      k (); bits.(i) ← Black; lf ()
    end else begin
      lf (); bits.(i) ← White; k ()
    end
```

Application à une forêt `f`

```
loop_f (fun () → (* affichage *)) f ()
```

Complexité

C_2 a une complexité $O(n)$ en espace

$O(N)$ en temps

C_2 est optimal

Performances dans un rapport satisfaisant avec le code *loopless* de Knuth ($<$ 2,5)

IV

Encore moins d'ordre supérieur

En collaboration avec François Pottier (INRIA Rocquencourt)

Dé-fonctionalisation

Technique consistant à remplacer les fermetures d'ocaml par une structure spécialisée (Danvy & Nielsen, Cejtin *et al.*)

```
type continuation =
  | Display
  | Continue of int × continuation × continuation
```

C_3 = dé-fonctionalisation de C_2

```
val loop_f : continuation → forest → continuation
val loop_t : continuation → tree → continuation
```

```
let rec loop_f k = function
  | [] → k
  | t :: f → loop_t (loop_f k f) t
and loop_t k (Node (i,f)) =
  Continue (i, k, loop_f k f)
```

C_3 (suite)

```
let rec run = function
  | Display →
      (* affichage *)
  | Continue (i, k, lf) →
      if bits.(i) = White then begin
        run k; bits.(i) ← Black; run lf
      end else begin
        run lf; bits.(i) ← White; run k
      end

run (loop_f Display f)
```

Résultat de la dé-fonctionalisation

Gain : entre 20 et 30 % par rapport à C_2

Ultime raffinement

Exactement une continuation par nœud

⇒ la continuation est l'indice du nœud

Les continuations associées `k` et `lf` dans deux tableaux

```
val ka  : int array
val lfa : int array
```

La continuation initiale représentée par `-1`

Dernier code C_4

```
val loop_f : int → forest → int
val loop_t : int → tree → int

let rec loop_f k = function
  | [] → k
  | t :: f → loop_t (loop_f k f) t
and loop_t k (Node (i,f)) =
  ka.(i) ← k;
  lfa.(i) ← loop_f k f;
  i
```

C_4 (suite)

```
let rec run = function
  | (-1) →
      (* affichage *)
  | i →
      if bits.(i) = White then begin
        run ka.(i); bits.(i) ← Black; run lfa.(i)
      end else begin
        run lfa.(i); bits.(i) ← White; run ka.(i)
      end

run (loop_f (-1) f)
```

Résultats

Maintenant aussi performant que le code *loopless* de Knuth

(meilleur sur certaines forêts, moins bon sur d'autres)

Le code reste concis, n'alloue plus dynamiquement

Les informations dans `ka` et `lfa` peuvent être interprétées simplement, comme des relations entre les nœuds

V

Travail en cours et perspectives

Preuve formelle

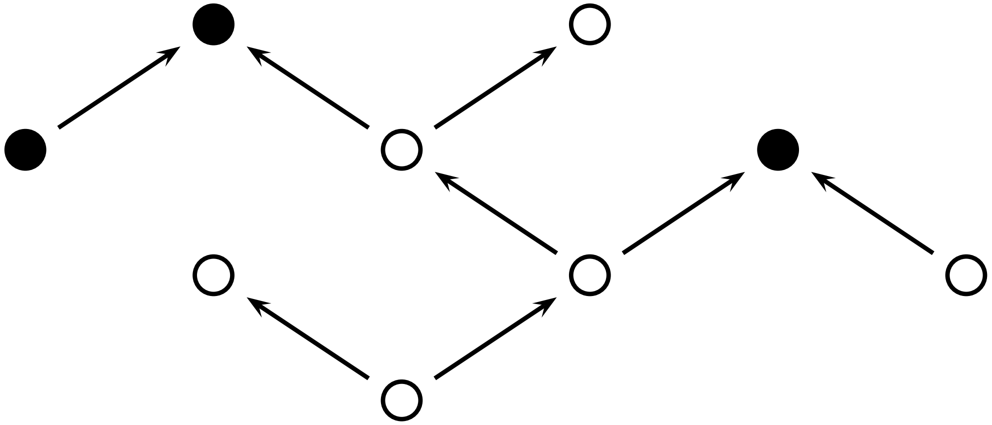
Une preuve formelle de C_1 en cours, avec Coq

Traits impératifs + ordre supérieur : une première ?

Caractérisation inductive des configurations finales

Généralisation

Algorithme de Li-Ruskey : généralisation à des graphes orientés totalement acycliques (non publié)



Continuations + graphes inductifs (M. Erwig) ?