Backtracking

Quentin Fortier

March 26, 2024

Backtracking

Définition

Le **backtracking** (ou : **retour sur trace**) consiste à construire une solution petit à petit, en revenant en arrière s'il n'est pas possible de l'étendre en une solution complète.

Backtracking

Définition

Le **backtracking** (ou : **retour sur trace**) consiste à construire une solution petit à petit, en revenant en arrière s'il n'est pas possible de l'étendre en une solution complète.

Exemples :

Backtracking

Définition

Le **backtracking** (ou : **retour sur trace**) consiste à construire une solution petit à petit, en revenant en arrière s'il n'est pas possible de l'étendre en une solution complète.

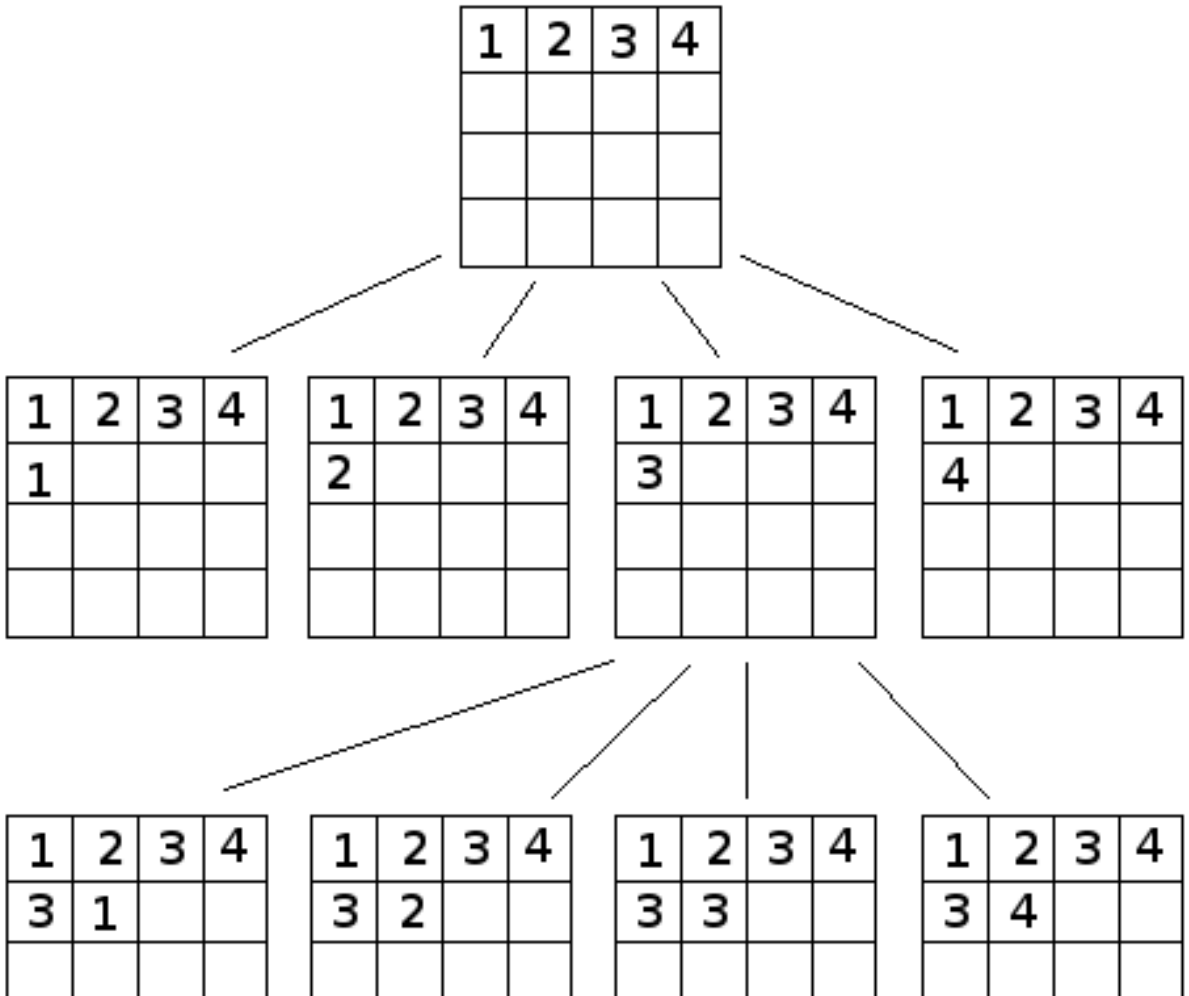
Exemples :

- Résolution d'un sudoku en le remplissant case par case.
  S'il n'est pas possible de mettre un numéro dans une case, on revient en arrière sur le dernier choix effectué pour en choisir un autre.
- Coloriage d'un graphe avec k couleurs de façon à ce que 2 sommets adjacents soient de couleurs différentes.

Souvent, cela donne une complexité exponentielle.

Backtracking

Un backtracking revient à faire un DFS sur l'arbre des possibilités, en passant à une branche suivante lorsqu'il n'est pas possible d'étendre une solution partielle :



Exemple : Sudoku

Résolution d'un sudoku par backtracking en Python :

```
def solve_sudoku(m):
    # m[i][j] = 0 si la case (i, j) est vide
    def aux(i, j): # (i, j) = case courante
        if i == 9:
            return True
        if j == 9:
            return aux(i+1, 0)
        if m[i][j] != 0:
            return aux(i, j+1)
        for k in range(1, 10):
            if is_valid(i, j, k):
                m[i][j] = k
                if aux(i, j+1):
                    return True
                m[i][j] = 0
        return False
    return aux(0, 0)
```

Exemple : Algorithme de Quine

Pour résoudre SAT, on a vu la méthode « brute force » consistant à tester les 2^n possibilités pour les n variables.

Exemple : Algorithme de Quine

Pour résoudre SAT, on a vu la méthode « brute force » consistant à tester les 2^n possibilités pour les n variables.

L'**algorithme de Quine**, par backtracking, est plus efficace (mais toujours en complexité exponentielle). Il suppose que la formule est en FNC :

```
type litteral = V of int | NV of int
type cnf = litteral list list
```

Exemple : Algorithme de Quine

Pour résoudre SAT, on a vu la méthode « brute force » consistant à tester les 2^n possibilités pour les n variables.

L'**algorithme de Quine**, par backtracking, est plus efficace (mais toujours en complexité exponentielle). Il suppose que la formule est en FNC :

```
type litteral = V of int | NV of int
type cnf = litteral list list
```

Idée :

1. Prendre une variable x restante dans la formule
2. Tester récursivement si $f[x \leftarrow T]$ est satisfiable
3. Si non, tester récursivement si $f[x \leftarrow F]$ est satisfiable

Exemple : Algorithme de Quine

Dans $f[x \leftarrow T]$, on effectue des simplifications :

- Si une clause contient x, on enlève cette clause (elle est vraie)
- Si une clause contient $\neg x$, on enlève $\neg x$ de cette clause (ce littéral ne peut pas être vrai)
- Si une clause devient vide, la formule est fausse (on backtrack)
- Si une formule contient aucune clause, elle est vraie (on a trouvé une solution)

Exemple : Algorithme de Quine

```
let var x b = if b then V x else NV x

(* subst f x b calcule f[x <- b] et simplifie *)
(* renvoie None si on obtient F, f[x <- b] sinon *)
let rec subst f x b = match f with
  | [] -> Some []
  | c::q -> let c = List.filter ((<>) (var x (not b))) c in
    match subst q x b with
      | None -> None
      | Some s ->
        if c = [] then None
        else if List.mem (var x b) c then Some s
        else Some (c::s);;
```

Exemple : Algorithme de Quine

```
let var x b = if b then V x else NV x

(* subst f x b calcule f[x <- b] et simplifie *)
(* renvoie None si on obtient F, f[x <- b] sinon *)
let rec subst f x b = match f with
  | [] -> Some []
  | c::q -> let c = List.filter ((<>) (var x (not b))) c in
    match subst q x b with
      | None -> None
      | Some s ->
        if c = [] then None
        else if List.mem (var x b) c then Some s
        else Some (c::s);;
```

Ceci permet parfois de se rendre compte plus tôt que la formule n'est pas satisfiable, donc d'énumérer moins de cas.

Exemple : Algorithme de Quine

```
let rec get_var = function
  | ((V x)::_)::_ | ((NV x)::_)::_ -> x
  | _ -> failwith "get_var"

let rec quine f =
  if f = [] then true
  else
    let x = get_var f in
    List.exists (fun v -> match subst f x v with
      | Some s -> quine s
      | None -> false) [false; true];
```