Backtracking et algorithme de Quine

Quentin Fortier

April 22, 2022

Backtracking

Définition

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

Backtracking

Définition

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

Exemples :

Backtracking

Définition

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

Exemples :

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

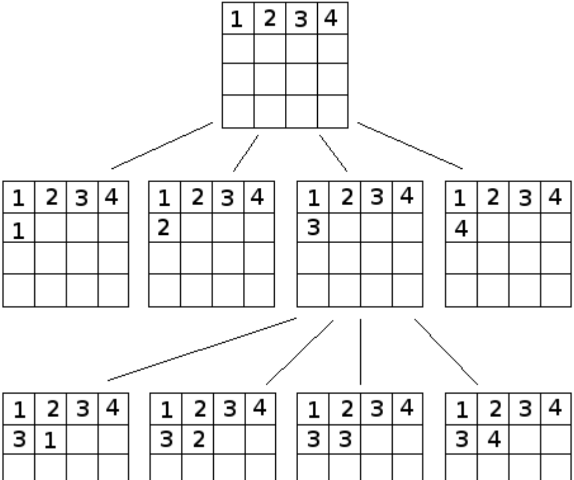
Backtracking

Résolution d'un sudoku par backtracking, en utilisant un entier en base 2 pour stocker l'ensemble des valeurs possibles sur une case :

```
bool backtrack(int m[9][9], int i, int j) {
    if(i > 8)
        return true;
    if(m[i][j] != -1)
        return backtrack(m, i + j/8, (j + 1)%9);
    int f = line(m, i) | column(m, j) | square(m, 3*(i/3) + j/3);
    for(int k = 0; k < 9; k++)
        if(f & to_set(k) == 0) {
            m[i][j] = k;
            if(backtrack(m, i + j/8, (j + 1) % 9))
                return true;
            m[i][j] = -1;
        }
    return false;
}
```

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 :



Algorithme de Quine

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

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. Il suppose que la formule est en FNC :

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

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. 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

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)

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);;
```

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.

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];
```

Algorithme de Quine

Il est possible d'utiliser des **heuristiques** pour guider la recherche améliorer les performances du backtracking, dans certains cas.
Par exemple, on peut choisir le littéral le plus fréquent à chaque étape (qui permettra de simplifier davantage).

Algorithme de Quine

Extensions de l'algorithme de Quine (HP) :

- **Algorithme DPLL** : Lorsqu'il n'y a plus qu'un seul littéral dans une clause, sa valeur est fixée et peut être propagée.
- **Algorithme CDCL** : Si on est bloqué après avoir mis des variables x, y, z à T, on ajoute la clause $\neg x \vee \neg y \vee \neg z$ (**apprentissage de clause**). On espère ainsi pouvoir éliminer plus de branches dans la recherche ultérieure dans l'arbre d'exploration.

Algorithme de Quine

Extensions de l'algorithme de Quine (HP) :

- **Algorithme DPLL** : Lorsqu'il n'y a plus qu'un seul littéral dans une clause, sa valeur est fixée et peut être propagée.
- **Algorithme CDCL** : Si on est bloqué après avoir mis des variables x, y, z à T, on ajoute la clause $\neg x \vee \neg y \vee \neg z$ (**apprentissage de clause**). On espère ainsi pouvoir éliminer plus de branches dans la recherche ultérieure dans l'arbre d'exploration.

Exercice

À faire pendant les vacances : comparer la vitesse d'exécution de différentes méthodes de résolution de SAT (brute force, Quine, avec heuristiques...). On pourra s'inspirer du code du TP de benchmark pour la recherche de sous-mot.