Recherche par force brute

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Principe

Problème de décision et exploration exhaustive

Problème

Considérons un problème du type “trouver $x \in V$ vérifiant une propriété $P(x)$”.

Exemple

- V est l'ensemble des chaînes de caractères, et $P(x)$ vérifie si $x \in V$ est le mot de passe que l'on cherche ;
- V est l'ensemble des indices possibles dans un tableau, et $P(i)$ vérifie si $t[i] = v$, où v est une valeur que l'on cherche ;
- V est l'ensemble des grilles complétées d'un problème de Sudoku, et P vérifie si la grille est valide ;
- V est l'ensemble d'assemblages de pièces d'un puzzle, et P vérifie si le puzzle est correct, i.e. si deux pièces côte à côte ont des côtés compatibles.

Problème de décision et exploration exhaustive

Remarque

Dans certains problèmes, un tel x n'est pas unique, et on cherche à tous les énumérer.

Brute Force

Une **recherche par force brute** ou **recherche exhaustive** consiste à énumérer l'ensemble V jusqu'à obtenir une solution en testant P pour chaque valeur rencontrée.

Problème de décision et exploration exhaustive

```ocaml
exception Trouve of int ;;

let recherche t x =
  try
    for i = 0 to Array.length t - 1 do
      if t.(i) = x
      then raise (Trouve i)
    done;
    raise Not_found
  with Trouve i -> i
;;
```

Exemple

Parmi les problèmes précédents, la recherche linéaire dans un tableau est le plus simple, et le programme ci-dessus est caractéristique d'une recherche exhaustive.

Problème de décision et exploration exhaustive

Algorithme 1 : Algorithme de recherche exhaustive

pour $v \in V$ **faire**
  si $P(v)$ *est vérifié* **alors**
    s'arrêter avec la solution v

Principe

La forme usuelle d'un algorithme de recherche exhaustive sera donc comme ci-dessus.

On rappelle que pour pouvoir s'arrêter au cours de l'énumération en OCaml, si on programme en impératif, on utilise en général des **exceptions**.

Problème de décision et exploration exhaustive

Énumération

Cela pose naturellement la question de l'énumération des éléments de V.

Si c'est immédiat dans l'exemple peu pertinent de la recherche dans un tableau, c'est beaucoup plus complexe pour l'énumération des assemblages d'un puzzle par exemple.

Exemple

Pour la recherche du mot de passe, on pourrait commencer par énumérer les chaînes de longueur 1, puis de longueur 2, et ainsi de suite.

Problème de décision et exploration exhaustive

Complexité

Le plus souvent, l'ensemble V est fini (pour les mots de passe, cela peut constituer à limiter la longueur maximale du mot de passe).

Ainsi, une recherche exhaustive par force brute effectue $\mathcal{O}(|V|)$ itérations.

Attention

Cela ne veut pas dire que la complexité de l'algorithme est en $\mathcal{O}(|V|)$, car tester $P(v)$ peut être coûteux.

Problème d'optimisation et exploration exhaustive

Problème d'optimisation

On retrouve la notion d'exploration exhaustive ou force brute pour les problèmes d'optimisation. Il s'agit de problèmes de la forme : déterminer $x \in V$ tel que $f(x)$ soit minimale ou maximale.

L'exploration exhaustive consiste alors à calculer toutes les images par f des éléments de V afin de déterminer un extremum.

Recherche par retour sur trace (Backtracking)

Construction itérative de candidats

Construction itérative

Dans de nombreux cas, l'ensemble V peut se décrire par un processus itératif de construction de ces éléments.

Une manière de voir cela est de parler de positions et de mouvements, ou coups.

Construction itérative de candidats

Exemple

Considérons un puzzle comme le puzzle Eternity II, constitué de 256 pièces carrées. Une configuration finale du puzzle consiste à avoir placé les 256 pièces. Parmi celles-ci, les configurations valides sont celles satisfaisant les contraintes de chaque côté.

Comme pour tous les puzzles, la position initiale est un plateau vide, et chaque mouvement consiste à place une pièce disponible dans un emplacement disponible. Cela correspond à la manière dont on procèderait à la main.

Construction itérative de candidats

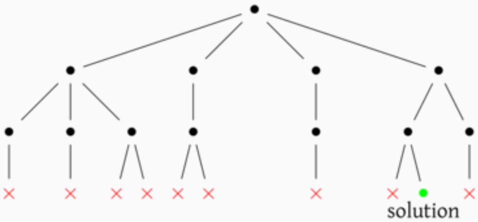
Exemple

Le puzzle Eternity II a été spécialement conçu pour être difficile à résoudre par force brute avec un ordinateur.

Le puzzle est sorti le 28 juillet 2007, et 2 millions de dollars étaient promis à la première personne qui trouverait une solution avant le 31 décembre 2010.

Personne n'a encore trouvé de solution.

Construction itérative de candidats



Arbres

Ainsi, on peut représenter la construction de V sous la forme d'un arbre dont les nœuds sont les positions et les arêtes les mouvements. Les positions complètes sont les feuilles de l'arbre, elles correspondent aux éléments de V et ce sont donc celles-ci qu'on va explorer pour y trouver une solution.

L'avantage de cette représentation est qu'elle découle naturellement d'un parcours récursif des positions.

Le type option en OCaml

Solution partielle

Nous allons être confronté à la situation suivante : écrire une fonction qui parfois renvoie une valeur, et parfois rien.

Pour symboliser l'absence de valeur dans certains langages comme le C, nous utiliserions une valeur spéciale comme -1 ou `NULL` par exemple.

En OCaml, le système de type nous permet de faire mieux.

Le type option en OCaml

```
type 'a option =
  | None
  | Some of 'a
;;
```

Type option

Le type `'a option`, déjà définit comme ci-dessus par OCaml, permet de représenter soit une valeur de type `'a` soit une absence de valeur.

Le type option en OCaml

```ocaml
match o with
  | None -> (* ... *)
  | Some a -> (* ici on peut accéder au contenu a *)
```

Type option

Pour manipuler une valeur `o` de type `'a option`, on utilisera un filtrage.

Le type option en OCaml

Exemple

Pour représenter une grille de Sudoku partiellement remplie, on pourra utiliser un `int option array array`.

Recherche par retour sur trace (backtracking)

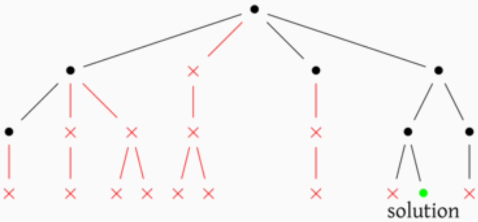
Backtracking

Si on reprend la construction itérative précédente, on se rend compte qu'elle n'est pas très intelligente : faut-il remplir l'intégralité d'un puzzle avant de se rendre compte qu'il est invalide en raison des deux premières pièces ?

On peut donc raffiner l'approche précédente en introduisant une notion de mouvements valides qui sont les mouvements qui préservent la correction partielle.

Cette amélioration de la recherche exhaustive s'appelle la **recherche par retour sur trace** (**backtracking** en anglais).

Recherche par retour sur trace (backtracking)



Backtracking

C'est un changement simple qui peut avoir un grand impact sur l'arbre des positions.

En reprenant l'illustration précédente, on peut imaginer que cela reviendrait à ne pas parcourir les mouvements rouges et les sous-arbres associés.

Exemple : résolution de Sudoku

Sudoku

la recherche par retour sur trace se prête très bien à la résolution de problèmes comme le Sudoku.

On va ici tout simplement tenter de remplir chaque case du haut vers le bas tant qu'on satisfait les contraintes du Sudoku.

Exemple : résolution de Sudoku

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

Exemple : résolution de Sudoku

Sudoku

Commençons par rappeler le principe du Sudoku :

- on part d'une grille de 81 cases réparties en une grille de 3×3 sous-grilles de 3×3 cases, et comportant des chiffres de 1 à 9 dans certaines cases ;
- l'objectif est de remplir chaque case avec un chiffre de 1 à 9 de sorte que chaque ligne, chaque colonne, et chaque sous-grille 3×3 comporte une et une seule fois chaque chiffre ;
- un Sudoku admet une unique solution.

Exemple : résolution de Sudoku

```
type grille = int option array array ;;
```

OCaml

Pour représenter une grille de Sudoku en OCaml, on utilise un tableau de type `int option array array`, la valeur `None` signifiant que la case est vide, et la valeur `Some x` qu'elle est remplie avec la valeur x.

Exemple : résolution de Sudoku

```
let probleme =
  [|
    [| Some 1; None; None; None; None; None; None; None; Some 6 |];
    [| None; None; Some 6; None; Some 2; None; Some 7; None; None |];
    [| Some 7; Some 8; Some 9; Some 4; Some 5; None; Some 1; None; Some 3 |];
    [| None; None; None; Some 8; None; Some 7; None; None; Some 4 |];
    [| None; None; None; None; Some 3; None; None; None; None |];
    [| None; Some 9; None; None; None; Some 4; Some 2; None; Some 1 |];
    [| Some 3; Some 1; Some 2; Some 9; Some 7; None; None; Some 4; None |];
    [| None; Some 4; None; None; Some 1; Some 2; None; Some 7; Some 8 |];
    [| Some 9; None; Some 8; None; None; None; None; None; None |];
  |] ;;
```

Exemple

La variable `probleme` ci-dessus code le Sudoku de l'exemple précédent.

Exemple : résolution de Sudoku

```
let rec suivant g (x,y) =
  if x > 8 then None
  else if g.(x).(y) = None then Some (x,y)
  else if y < 8 then suivant g (x, y+1)
  else suivant g (x+1, 0)
;;
```

Suivant

On commence par définir une fonction `suivant` telle que `suivant g (x,y)` renvoie **`Some`** `(xi,yi)` quand (x_i, y_i) sont les coordonnées de la prochaine case libre après (x, y), ou **`None`** quand il n'existe pas de telle case libre.
Cela signifie alors que la grille est entièrement remplie.

Exemple : résolution de Sudoku

```
let valide g (x,y) =
  let v = ref true in
  let vus_colonne = Array.make 9 false in
  for x' = 0 to 8 do
    match g.(x').(y) with
    | None -> ()
    | Some k -> if vus_colonne.(k-1) then v := false;
                vus_colonne.(k-1) <- true
  done;
  let vus_ligne = Array.make 9 false in
  for y' = 0 to 8 do
    match g.(x).(y') with
    | None -> ()
    | Some k -> if vus_ligne.(k-1) then v := false;
                vus_ligne.(k-1) <- true
  done;
  let vus_grille = Array.make 9 false in
  let xb = (x / 3) * 3 in
  let yb = (y / 3) * 3 in
  for xd = 0 to 2 do
    for yd = 0 to 2 do
      match g.(xb+xd).(yb+yd) with
      | None -> ()
      | Some k -> if vus_grille.(k-1) then v := false;
                  vus_grille.(k-1) <- true
    done
  done;
  !v ;;
```

Valide

On définit également une fonction `valide` telle que `valide g x y` renvoie `true` si et seulement si la valeur placée en (x, y) n'invalide pas la grille.

Exemple : résolution de Sudoku

```
exception Solution ;;

let resout g =
  let rec aux xi yi = match suivant g (xi, yi) with
    | None -> raise Solution
    | Some (x,y) ->
      for i = 1 to 9 do
        g.(x).(y) <- Some i;
        if valide g (x,y)
        then begin
          aux x y
        end
      done;
      g.(x).(y) <- None
  in
  try
    aux 0 0
  with Solution -> ()
;;
```

Résolution

On peut alors définir une fonction `resout` qui va résoudre le Sudoku en effectuant tous les remplissages **tant qu'on a une grille valide**.

Exemple : résolution de Sudoku

```
exception Solution ;;

let resout g =
  let rec aux xi yi = match suivant g (xi, yi) with
    | None -> raise Solution
    | Some (x,y) ->
      for i = 1 to 9 do
        g.(x).(y) <- Some i;
        if valide g (x,y)
        then begin
          aux x y
        end
      done;
      g.(x).(y) <- None
  in
  try
    aux 0 0
  with Solution -> ()
;;
```

Résolution

Dès qu'une solution est trouvée, on s'arrête. Pour cela, on utilise le mécanisme des **exceptions** pour permettre une sortie prématurée.

Exemple : résolution de Sudoku

```
exception Solution ;;

let resout g =
  let rec aux xi yi = match suivant g (xi, yi) with
    | None -> raise Solution
    | Some (x,y) ->
      for i = 1 to 9 do
        g.(x).(y) <- Some i;
        if valide g (x,y)
        then begin
          aux x y
        end
      done;
      g.(x).(y) <- None
  in
  try
    aux 0 0
  with Solution -> ()
;;
```

Résolution

On fait le choix de travailler **en place** dans la grille. Ainsi, à la fin de l'exécution de la fonction, la grille correspond à la solution.

Exemple : résolution de Sudoku

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

Exemple

Une vidéo fournie en annexe montre l'évolution de l'algorithme sur l'exemple ci-dessus.

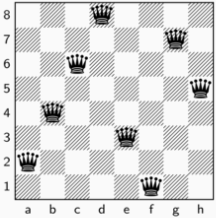
Énumération de toutes les solutions

Énumération de toutes les solutions

Énumération

Le problème précédent du Sudoku n'avait par définition qu'une unique solution. Cependant, il existe des problèmes pour lesquels plusieurs solutions existent, et pour lesquels on souhaite les énumérer.

Exemple : problème des huit reines



Problème des huit reines

L'exemple classique de ce problème est celui des **huit reines** : étant donné un échiquier, peut-on place huit reines de sorte qu'aucune reine ne puisse prendre une autre reine ?

Plus précisément, sur un plateau de 8×8 cases, peut-on placer huit pions tels que deux poins quelconques ne soient jamais sur la même ligne, la même colonne, ou la même diagonale ?

Exemple : problème des huit reines

Solutions partielles

Ce problème admet effectivement des solutions partielles, en ne considérant que les k premières reines à placer.

Pour énumérer les solutions, on peut même se contenter de solutions partielles où les k reines sont placées sur les k premières rangées.

Exemple : problème des huit reines

Algorithme

Voici un algorithme pour énumérer les solutions :

- Supposons que k reines aient été placées, et qu'on dispose d'une solution partielle.
  - Si $k = 8$, alors toutes les reines sont placées, et la solution est complète. On la comptabilise.
  - Sinon, on continue la recherche pour chaque position de la $k + 1$-ème reine sur la $k + 1$-ème rangée qui préserve le fait d'être une solution partielle.

Ici, quand on dit qu'on continue la recherche, ce qu'on signifie, c'est qu'on effectue un appel récursif.

Exemple : problème des huit reines

```
let rec valide (x,y) l = match l with
  | [] -> true
  | (x',y')::q -> x <> x' && (* lignes différentes *)
                  y <> y' && (* colonnes différentes *)
                  abs(x'-x) <> abs(y'-y) && (* diagonales différentes *)
                  valide (x,y) q
;;
```

Valide

La fonction `valide` vérifie si placer une reine en (x, y) est valide (la liste `l` contient les positions des autres reines).

Exemple : problème des huit reines

```
val resout_reines : (int * int) list -> (int * int) list list = <fun>
```

Résolution

On va maintenant définir une fonction récursive `resout_reines` telle que `resout_reines part` renvoie la liste des solutions complètes construites à partir des solutions partielles `part`.

Exemple : problème des huit reines

```
let rec resout_reines part =
  let k = List.length part in
  if k = 8 then [ part ]
  else
    begin
      let resultats = ref [] in
      for y = 0 to 7 do
        let essai = (k,y)::part in
        if valide (k,y) part
        then begin
          resultats := (resout_reines essai) @ !resultats;
        end
      done;
      !resultats
    end
;;
```

Résolution

Voici une implémentation où l'on explore les solutions à l'aide d'une boucle **for** dans l'appel récursif.

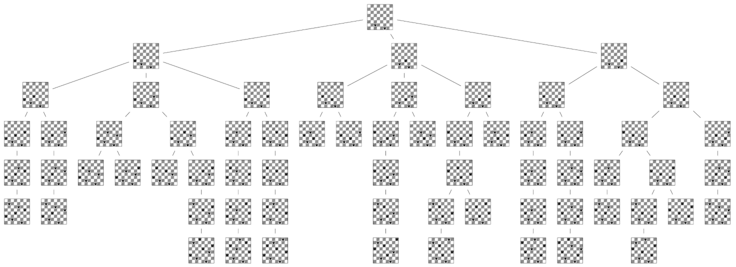
Exemple : problème des huit reines

```
let rec resout_reines part =
  let k = List.length part in
  if k = 8 then [ part ]
  else
    let rec aux y acc = match y < 0 with
      | true -> acc
      | false ->
        let essai = (k,y)::part in
        let acc' = if valide (k,y) part
          then (resout_reines essai) @ acc
          else acc in
        aux (y-1) acc'
    in
    aux 7 []
;;
```

Résolution

Voici une implémentation purement récursive, à l'aide d'une fonction auxiliaire récursive.

Exemple : problème des huit reines



Arbre de recherche

Une partie de l'arbre de recherche est représenté ci-dessus.

Le fichier image est fourni en annexe si vous voulez zoomer.

Exemple : problème des huit reines

Arbre de recherche

L'arbre complet comporte 2057 nœuds, dont 92 feuilles correspondent aux solutions du problème.

À titre de comparaison, l'arbre exhaustif correspondant à faire tous les choix de placements (en se limitant à une reine par ligne) compterait $8^8 = 16777216$ nœuds.

On voit donc bien que le **backtracking** est plus économe en exploration que le **brute force**.