Chapitre 2

Une utilisation de la récursivité : le backtracking

Le backtracking aussi appelé recherche avec retour-arrière ou recherche par essais-erreurs, permet de résoudre bon nombre de problèmes combinatoirs pour lesquels on ne dispose pas d'algorithme direct. L'idée est de procéder par **essais successifs**, chaque échec remettant en question les choix antérieurs.

On construit donc progressivement une solution possible par extension d'une solution partielle.

Un exemple très classique est le problème des huit reines[1]. Il s'agit de placer huit reines sur un échiquier (généralement 8*8) de sorte qu'aucune ne puisse en prendre une autre (*i.e.* que deux reines ne soientt jamais ni sur la même ligne ni sur la même colonne ni sur la même diagonale). On commence donc par choisir une case sur la premiere ligne et par y placer une reine. Ensuit on choisit parmi les cases possibles sur la deuxième ligne où placer la seconde reine, si à un moment donné il est impossible de placer une nouvelle reine sans qu'elle soit en prise on retire la reine placée à l'étape précédente et on essaye une autre case etc. (Essayez première colonne de la première ligne et troisième de la seconde et constatez qu'il vous sera impossible de finir de placer les autres...Ok à la main c'est un peu rebarbartif....)

Un autre grand classique est le jeu "le compte est bon". Il consiste à choisir un nombre entier positif `n`, une liste d'entiers positifs et d'extraire de la liste une sous liste de nombres dont la somme soit égale à `n`. On donc choisit récursivement dans la liste un nombre `x` inférieur à `n` et on essaye d'extraire de la liste restante les nombres dont la somme est égale à `n-x`. Lorsque cela n'est pas possible (il y a échec), on remet en cause le choix de `x`.

Dans tous les cas, on a un arbre des états possibles dont la racine est l'état initial (dans l'exemple des huit reines, aucune reine sur l'échiquier) et les fils de chaque nœud sont les états suivants (dans l'exemple la racine a huit fils représentant les huit cases possibles sur la premiere ligne. Si le premier de ces fils correspond à l'état où il y a une reine sur la première case de la première ligne, il a lui-même 6 fils car les cases un et deux de la seconde ligne ne sont plus admissibles). Certaines des branches de l'arbre conduisent à des echecs, d'autres mènent à des solutions.

La recherche d'une solution se résume donc à un parcours de l'arbre des états. Les différents parcours correspondent à différentes stratégies. Dans le cas des huit reines, un parcours "en profondeur d'abord" est évidemment plus efficace, c'est généralement le cas et c'est donc ce parcours que nous mettrons en œuvre dans nos exemples. Néanmoins, il peut arriver que le parcours en largeur d'abord soit

[1] posé par Gauss en 1850

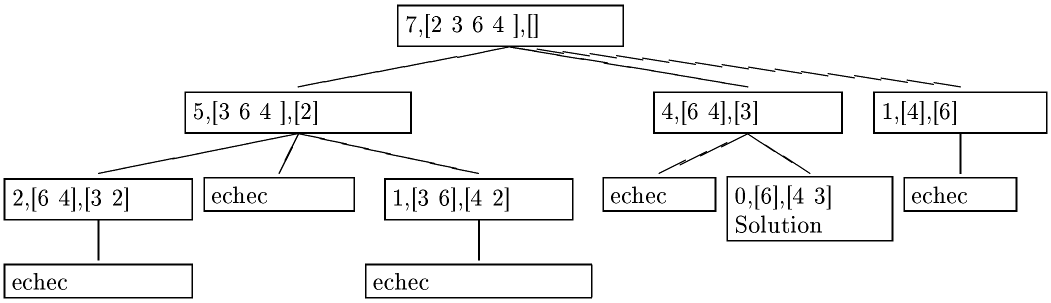

Fig. 2.1. Une partie d'un arbre d'états du compte est bon

plus efficace (dans le cas du compte est bon, si la liste contient le nombre cherché il ne nécessite qu'une étape).

On définit une fonction `uneSolution` qui à un état de l'exploration associe une solution prolongeant cet état et lève l'exception `Echec` si il n'y a pas de telle solution. Le principe est le suivant :

1. Si on est dans un état final (`i.e.` sans suivant), soit c'est une solution et on la retourne, soit c'est un échec et on lève l'exception.
2. Si on est dans un état intermédiaire, on fait la liste de tous les états suivants et on lance une recherche à partir de chacun d'eux jusqu'à ce que l'on trouve une solution.

On suppose donnés les prédicats `estFinal` et `estSolution` qui testent respectivement si un état est final et s' il est solution, et la fonction `lEtatsSuivant` qui associe à un état la liste des états suivants. La fonction de recherche d'une solution s'écrit alors:

```
let  rec uneSolution etat =
  if (estFinal etat) then
     if (estSolution etat) then etat else raise Echec
  else (some uneSolution (lEtatsSuivant etat) Echec);;
```

(Les fonction `estFinal`, `estSolution` et `lEtatsSuivant` seront évidemment des paramètres nous les avons introduis séparément pour des questions de lisibilité.)

2.1 Application

Le problème des huit reines

On place une reine sur la première ligne à un case donnée. puis on va ajouter récursivement les suivantes sur les ligne suivante de manière à ce que chaque reine ajoutée ne soit pas en prise par les précédentes (cela est vérifié par `estAdmissible`). A chaque étape,un état est une liste d'entiers représentant un numéro de colonne. Le numéro de la ligne correspondant est la position dans la liste.

```
let rec iota n k = if n>k then [] else n::(iota (n+1) k);;

let rec some f l e= match l with
 []->raise e
 |x::xs-> try (f x) with e->(some f xs e);;

(* l'etat est final si on a place huit reines *)
let estFinal etat = (list_length etat)=8;;
(* il est aussi solution ! *)
let estSolution =estFinal;;


(* test si on peut ajouter une reine nouvelleReine sur la ligne suivante *)
(* lorsque l'on est dans l'etat lReine *)

let estAdmissible nouvelleReine lReine=
     let rec aux lreine distance =match lreine  with
        []->true
        |x::xs->((not (x=nouvelleReine+distance)) &
                 (not (x=nouvelleReine)) &
                 (not (x=nouvelleReine-distance)) &
                 (aux xs (1+distance))) in
      aux lReine 1;;


let lEtatsSuivant etat =
 flat_map (fun position -> if (estAdmissible position etat) then
                              [position::etat] else []) (iota 1 8);;

exception Echec;;

let  rec uneSolution etat =
  if (estFinal etat) then
     if (estSolution etat) then etat else raise Echec
  else (some uneSolution (lEtatsSuivant etat) Echec);;

(*
#uneSolution [];;



- : int list = [4; 2; 7; 3; 6; 8; 5; 1]
*)
```

Le compte est bon

Un état est un triplet `(n,l,r)` où `n` est le nombre que l'on cherche à construire, `l` la liste des nombres disponible et `r`, la liste des nombres déjà retenu. Ici état final est solution ne sont pas identiques. Un état est final si on a utisé tout les nombres à notre disposition c'est-à-dire si `l` est vide. Il est solution si on a formé le nombre c'est-à-dire si `n` vaut zéro.

```
let estFinal (n,l,r) =  l=[];;
let estSolution (n,l,r) =n=0 ;;

let rec lEtatsSuivant (a,l1,l2)  = match l1 with
               []->[]
              |x::xs ->let d=a-x in if d>=0 then
                   (d,xs,x::l2)::(lEtatsSuivant (a,xs,l2)) else
                   (lEtatsSuivant(a,xs,l2));;


let  rec uneSolution etat =
  if (estFinal etat) then
     if (estSolution etat) then etat else raise Echec
  else (some uneSolution (lEtatsSuivant etat) Echec);;

(* exemples
uneSolution (29,[1;5;3;5;7;10;12],[]);;
 int * int list * int list = 0, [], [12; 7; 5; 5]
uneSolution (21,[1;1;3;5;7;10;12;15],[]);;
- : int * int list * int list = 0, [], [15; 5; 1]
*)
```

Le problème des mariages stables

C'est un autre grand classique. Imaginez-vous à la tête d'une agence matrimoniale. Étant donnés `n` hommes et `n` femmes et pour chacun d'eux la liste de ses préférences parmi les individus du sexe opposé, vous devez former `n` couples stables. Les instabilitées apparaissent quand dans deux couples (h_i, f_i) et (h_j, f_j), h_i préfère f_j à f_i et f_j préfère h_i à h_j. Pour rechercher une solution, on part d'un couple donné et on ajoute recursivement des couples en veillant à ne pas former d'instabilité. Si on ne peut plus ajouter de couple sans déstabiliser l'ensemble, on revient sur les choix précédents.

Pour en savoir plus sur ce beau problème et sur la stabilité des ménages vous entourant :

`\http://cartan.u-strasbg.fr:80/%7Eslc/opapers/s23dumont.html`