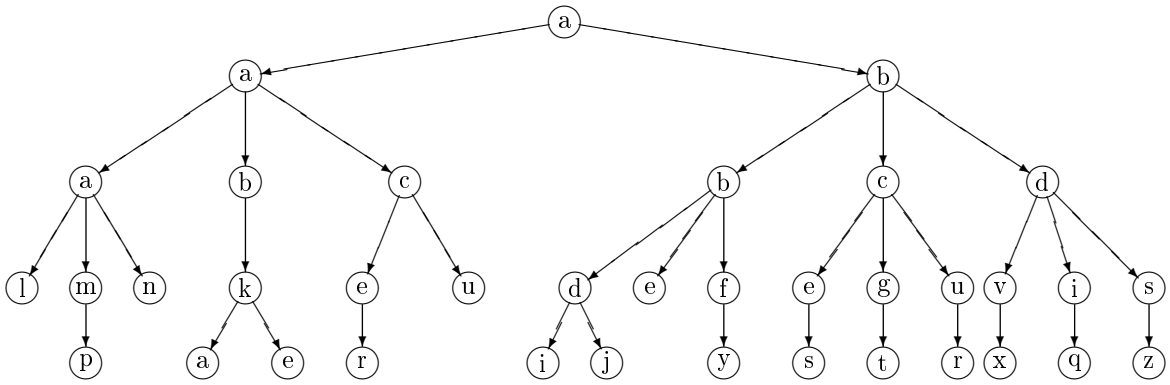
Retour sur trace (backtracking)

Introduction

Le retour sur trace (*backtracking* en V.O.) est une stratégie de résolution de problèmes dans lequels plusieurs contraintes doivent être simultanément vérifiées : on explore en profondeur l'arbre des possibles en faisant marche arrière dans le parcours à chaque fois qu'une incompatibilité apparaît.

Considérons par exemple le problème suivant : on dispose d'un arbre étiquetté par des lettres ; peut-on obtenir comme branche de cet arbre un mot de la langue française ? On ne va pas explorer toutes les branches ! On sait par exemple qu'aucun mot de la langue française ne commence par aa ou par abbd, il est inutile d'explorer jusqu'au bout une branche qui commencerait ainsi.



Complétons cette figure en indiquant les chemins explorés.

Si on interrompt l'exploration dès la première solution trouvée, on s'arrête une fois trouvé *abbé*. Sinon, on trouve une seule autre solution *abcès*.

I Un premier exemple : le problème des n reines

Aux échecs, une reine menace une autre pièce si cette autre pièce se trouve sur une même ligne, une même colonne, ou une même diagonale. On souhaite placer autant de reines que possibles sur un échiquier $n \times n$ sans qu'aucune n'en menace une autre. On peut clairement placer au plus une reine par ligne, donc on ne pourra pas placer plus de n reines. Est-il possible d'en placer n ? Si oui, comment trouver une solution, voire toutes les solutions ?

Avant d'examiner comment résoudre ce problème par backtracking, un peu de modélisation.
On code l'échiquier par une matrice $n \times n$, on localisera donc une pièce sur l'échiquier par ses coordonnées (i, j) avec $0 \leqslant i, j \leqslant n-1$ comme les matrices de OCaml (plutôt que A8, ..., H1 pour $n = 8$). Ci-dessous le détail pour $n = 4$:

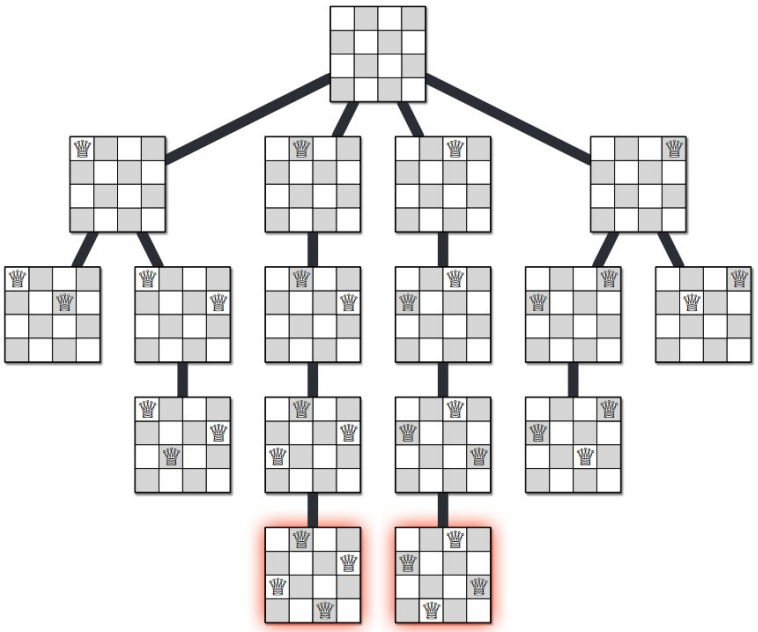
(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Commençons par écrire une fonction `menace : int*int → int*int → bool` telle que `menace (i,j) (k,l)` indique si une reine placéee en position (i, j) menace une pièce placée en position (k, l).

Imaginons maintenant avoir déjà placé k reines.
Comme on sait qu'une ligne ne peut contenir qu'une seule reine, on peut indiquer les positions de ces k reines comme suit : on utilise un tableau `t` tel que, pour i entre 0 et $k-1$, `t.(i)` donne l'entier j tel que la $i^{\text{ième}}$ reine est placée en (i, j).
Écrivons une fonction `placement_impossible : int array → int*int → bool` telle que `placement_impossible t (k,l)` indique si l'une des reines placées en positions `(i,t.(i))` pour $i < k$ menace la case (k, l).

On détaille maintenant la stratégie de résolution des n reines par backtracking : les n reines sont placées une par une successivement sur les n lignes. Pour placer la $k^{\text{ième}}$ reine, on choisit le premier emplacement libre de la $k^{\text{ième}}$ ligne s'il en existe un (sinon le placement échoue) : les branches précédentes de l'arbre des possibles ne seront pas explorées, ce serait inutile. Puis on effectue un appel récursif pour placer la $(k+1)^{\text{ième}}$ reine : si celui-ci échoue on passe à l'emplacement libre suivant (s'il existe), sinon l'appel récursif trouve une solution (le cas d'arrêt étant $k = n$).

Dans le cas $n = 4$, et en ne représentant que les chemins effectivements explorés, l'arbre des possibles se présente comme suit (l'illustration est tirée de *Algorithms* de Jeff Erickson) :



Écrire une fonction `reines : int → int array` telle que `reines n` retourne un tableau `t` de longueur `n` tel que les cases de coordonnées `(i,t.(i))` donnent des emplacements possibles pour placer les n reines sans qu'elles se menacent. Par exemple, pour $n = 4$, en explorant l'arbre des possibles dans le même ordre que sur l'exemple, la fonction doit retourner `[|1;3;0;2|]`.

Adapter la fonction pour obtenir `toutes_les_solutions : int → int array list` qui donne la liste de toutes les solutions, puis une fonction `nombre_de_solutions : int → int` qui donne le nombre de solutions. Il faut donc trouver 2 solutions pour $n = 4$ (et 92 pour $n = 8$).

II Autres exemples, notamment le Sudoku

Cf TP.