TD #15 — Backtracking (avec solutions)

Exercice 1. *I'm not going back!*

Quel est le problème posé par l'extrait de code suivant, à supposer qu'il emploie une approche par backtracking ? On suppose que `is_a_solution` (de spécifications évidentes) est correcte, et que si (l'objet pointé par) l'argument de `update_solution` peut être étendu en une nouvelle solution partielle, alors celle-ci le modifie en une telle solution et renvoie `true`, et sinon elle renvoie `false`.

```
bool update_solution(void *partial_solution);
bool has_a_solution(void *partial_solution) {
    if (is_a_solution(partial_solution)) {
        return true;
    }
    if (!update_solution(partial_solution)) {
        return false;
    }
    return has_a_solution(partial_solution);
}
```

Comme l'indique le titre de l'exercice, aucun retour en arrière n'est effectué par `has_a_solution`. Cette fonction risque ainsi de rater des solutions possibles si les solutions partielles peuvent (au moins parfois) être étendues de plusieurs façons.

Exercice 2. *Backtracking & persistance*

On s'intéresse à un extrait de programme OCaml imaginaire de résolution d'un problème imaginaire par backtracking. Cet extrait doit faire les choses suivantes : initialiser la case d'indice `i` d'un tableau `a` (supposée initialement valoir `None`) à `Some true` ; appeler une fonction `solve : (* snip *) -> bool` avec comme arguments `a` et `i+1`, qui s'évalue à `true` en cas de succès ; s'évaluer à `true` en cas de succès, sinon considérer l'alternative `Some false` pour la case d'indice `i`, et si nécessaire revenir en arrière en s'évaluant à `false`. Dans ce dernier cas, l'état « observable » de `a` après évaluation de l'extrait doit être le même qu'à son début.

1. Écrivez cet extrait de programme en supposant que le tableau `a` est représenté par un type persistant (« observationnellement immuable ») manipulé par une fonction `set : 'a parray -> int -> 'a -> 'a parray`

   On propose :
   ```
   (* snip *)
   solve (set a i (Some true))  (i + 1) ||
   solve (set a i (Some false)) (i + 1)
   (* snip *)
   ```
   Il n'y a rien à faire pour satisfaire la dernière contrainte : le type `'a parray` étant persistant, l'état observable de `a` ne peut pas changer.

2. De même pour une représentation par un type mutable avec une fonction `set : 'a marray -> int -> 'a -> unit`

   On propose :
   ```
   (* snip *)
   (set a i (Some true)  ; solve a (i + 1)) ||
   (set a i (Some false) ; solve a (i + 1)) ||
   (set a i None ; false)
   (* snip *)
   ```
   La dernière ligne est indispensable pour rétablir l'état initial de `a`.

Conseil : dans les deux cas, il peut être intéressant d'utiliser des fonctions booléennes paresseuses.

Exercice 3. *Problème des* ≤ 4 ♕

Le *problème des n* ♕ consiste à placer n ♕ (« dames » (ou « reines »)) sur un échiquier $n \times n$ de façon à ce qu'aucune pièce ne puisse en capturer une autre. (Autrement dit, aucune ligne, colonne ou diagonale de l'échiquier ne doit être occupée par plus d'une pièce.)
En utilisant une approche manuelle par backtracking :

1. Montrez qu'il n'existe pas de solution à ce problème pour $1 < n < 4$. (Utilisez les symétries disponibles !)
2. Trouvez une solution pour $n = 4$.

Exercice 4. *À propos de la détection de mauvaises solution partielles*

Un *adversaire* tire uniformément au hasard $s \in \{0,1\}^{128}$ et vous demande de le deviner. Vous pouvez soumettre (en une seule fois) une liste de *requêtes* de valeurs candidates, et pour chacune il répondra si cette valeur vaut s.

1. Montrez que $2^{128} - 1$ requêtes sont nécessaires pour toujours apprendre s.

   Soit q le nombre de requêtes, si $q < 2^{128} - 1$, il est possible que toutes les réponses de l'adversaire soient négatives et s peut alors valoir chacune des (au moins) $2^{128} - q > 1$ valeurs restantes.

2. Montrez que pour $q < 2^{128} - 1$ requêtes distinctes, la probabilité de succès (sur le tirage de s) est $q/2^{128}$.

   Cela suit immédiatement de la définition d'un tirage uniforme.

Vous pouvez maintenant soumettre de façon interactive des valeurs partielles $x \in \{0,1,?\}^{128}$, et l'adversaire dira si $x \subseteq s$ ou non.

3. Montrez que 128 requêtes (interactives) de ce type suffisent pour apprendre le secret s.

   128 requêtes de la forme $??? \cdots 0 \cdots ???$ avec un unique chiffre en $i$ème position pour $i \in [\![0, 127]\!]$ sont suffisantes pour apprendre un à un les 128 bits de s, et donc s tout entier.

Exercice 5. *In the end, I kind of ended up back where I was in the first place*

Un *jeu de taquin* est un jeu de puzzle typiquement constitué d'un rectangle de dimensions $n \times m$ rempli de $n \times m - 1$ petits carrés numérotés. L'état initial d'un puzzle est donné par la position des petits carrés, et une résolution consiste à décider s'il est possible d'ordonner les carrés d'une certaine façon (par exemple par numéros croissants dans un parcours en ligne, avec toutes les cases du rectangle occupées par un carré sauf celle la plus en bas à droite) uniquement en faisant glisser les petits carrés vers l'unique position sans carré.

1. Expliquez en quoi il n'est pas évident qu'un tel problème puisse se résoudre par backtracking ?

   Ce problème n'admet pas de structure évidente de solution partielle. En particulier, il n'est pas immédiatement clair comment l'on pourrait construire un arbre dont un parcours en profondeur énumérerait *une unique fois* toutes les solutions partielles.
   Comme dans l'histoire de Bollo, il y a un risque important de considérer plusieurs fois le même état du puzzle et donc de ne pas terminer (Naboo ne sera pas forcément là pour mettre un terme à l'aventure.)

2. Proposez une (éventuelle adaptation d'une) stratégie par backtracking qui permette de résoudre ce problème. Quel est l'impact de votre adaptation sur le coût de la résolution ?

   Une idée possible consiste à adapter le parcours en profondeur effectué par la stratégie de backtracking en mémorisant (par exemple dans une structure d'ensemble dynamique implémentée par un arbre binaire de recherche…) les configurations déjà rencontrées, ceci afin d'éviter de les visiter plusieurs fois. Les prochaines « solutions partielles » d'une configuration sont alors celles qui sont accessibles par un mouvement autorisé **et qui n'ont pas déjà été visitées**. Ainsi, on traitera une configuration dont toutes les « voisines » ont déjà été visitées de la même façon qu'une feuille dans une exploration d'arbre.
   Un surcoût évident de cette adaptation par rapport à une exploration d'arbre est qu'il est maintenant nécessaire de mémoriser les configurations visitées (possiblement en grand nombre), alors que cela n'était pas nécessaire auparavant.