Tentatives

Ce document est un corrigé de l'exercice sur les tentatives du TD5 du cours EA3 (Université Paris Diderot, 2014-2015).

Les cartes sont des couples d'entiers, stockés dans un tableau c à n cases ($n \leq 36$). Pour une carte $c[i] = (k, l)$, on appellera $entree(c[i])$ l'entier k et $sortie(c[i])$ l'entier l. On dira que deux cartes $c[i]$ et $c[j]$ sont *composables* (attention à l'ordre) si $sortie(c[i]) = entree(c[j])$.

On rappelle qu'une tentative *correcte* est une suite de cartes, représentée par une suite d'indices compris entre 1 et n, dans laquelle deux cartes adjacentes sont composables, et dans laquelle une carte du jeu c apparaît une fois au plus.

On veut énumérer les tentatives correctes. Pour cela, le backtrack est une bonne approche : en effet, si en énumérant les tentatives on ajoute une carte qui résulte en une tentative qui n'est pas correcte, aucun rajout ultérieur de cartes n'aboutira à une tentative correcte. En d'autres termes, tout le sous-arbre de l'arbre des tentatives (ordonnées par leur longueur) enraciné en cette tentative incorrecte est composé de tentatives incorrectes, et il est donc inutile de le parcourir.

Rappelons que c'est précisément dans ces situations que l'on utilise le backtrack : quand on énumère des objets qui, dès que l'on obtient une erreur, ne peuvent pas devenir corrects à nouveau si on poursuit l'énumération. Le backtrack consiste alors à laisser tomber l'énumération commençant en cet objet, et à revenir à des objets ayant une chance de produire un résultat correct.

Par exemple, dans le cas du Sudoku, on a utilisé le backtrack parce qu'un remplissage partiel faux ne peut pas donner une solution valide, même si l'on poursuit le remplissage de la grille. Pour les perles de Dijkstra, c'était pareil : en ajoutant de nouvelles perles, on ne peut pas enlever les motifs identiques adjacents qui rendent la combinaison invalide.

Par ailleurs, dans le backtrack, puisque l'on énumère des objets, et que l'on ne poursuit pas l'énumération quand il y a une erreur, il suffit de tester si une erreur apparaît *en tête de l'énumération*. Dans l'exemple des Sudoku, c'est pour

cela qu'il suffisait de vérifier que le remplissage d'une case n'ajoutait pas d'erreur à la grille. En effet, s'il y avait une erreur dans la grille, elle aurait été jetée par l'algorithme, et on ne serait plus en train d'essayer de la remplir. La situation était identique pour les perles de Dijkstra.

Revenons donc au cas des tentatives. On veut donc concevoir une méthode récursive (c'est beaucoup plus simple, cf. Sudoku) qui prend en argument une tentative supposée correcte, puis pour chaque carte de c, teste si son ajout à la tentative passée en argument donne une tentative correcte. Si c'est le cas, on l'affiche et on continue. Sinon, on ne fait rien avec cette nouvelle carte.

On va modéliser une tentative par un tableau t de longueur n d'entiers compris entre 0 et n. Il y aura un 0 dans une case si elle ne contient pas de carte (en particulier, il y aura des 0 à la fin du tableau représentant une tentative incomplète). A tout instant, on fera en sorte que t soit une tentative correcte, c'est-à-dire qu'un même entier k n'apparaisse pas à deux positions $i \neq j$, et que pour tout i dans $\{1, \cdots, n-1\}$, on ait

$$c[i] \neq 0 \text{ et } c[i+1] \neq 0 \implies c[i] \text{ et } c[i+1] \text{ sont composables}$$

On obtient ainsi la fonction `ajoutCorrect`(t, i, l) qui prend en argument une tentative t de longueur l et un entier i représentant l'indice d'une carte $c[i]$, et qui renvoie `vrai` si et seulement si l'ajout de la carte $c[i]$ à la fin de la tentative t (en position $l+1$ donc) donne une tentative correcte.

```
ajoutCorrect(t, i, l)
si  l == n  retourner faux
sinon si  l == 0  retourner vrai
fin si
pour  j  allant de  1  à  l  faire :
      si  t[j] == i retourner faux
      fin si
fin pour tout
retourner (sortie(t[l]) == entree(c[i]))
```

Dans le premier test, on regarde si la tentative t est déjà complète, auquel cas on répond `faux` puisqu'on ne peut rien rajouter, ou si elle est vide, auquel cas on répond `vrai`. Ensuite, on parcourt les cartes de la tentative t une à une, si on trouve déjà la carte $c[i]$ (c'est-à-dire si l'entier i est stocké dans la tentative t), on retourne faux. Si on finit la boucle sans retourner faux, c'est que la carte $c[i]$ n'est pas encore dans la tentative t. Dans ce cas, on regarde donc si la dernière carte de la tentative (à l'indice l) est composable avec la carte $c[i]$ qu'on voudrait insérer après elle.

On peut maintenant donner la fonction d'énumération `enumere`(t, l), qui prend en entrée une tentative *correcte* de longueur l, l'affiche (puisqu'elle est correcte), puis teste tout ajout de carte, et s'appelle sur toute tentative correcte

de longueur $l+1$ ainsi obtenue.

```
enumere(t, l)
afficher t
pour i allant de 1 à n faire :
      si ajoutCorrect(t, i, l)
            t[l + 1] ← i
            enumere(t, l + 1)
      fin si
fin pour tout
```

On appelle ensuite `enumere` sur le tableau vide de longueur 0 et sur l'entier $l = 0$ depuis une fonction `main`.