université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

PFA

L3 Info, L3 Mag Info

TP 11

Présentation

Le but du TP est de se familiariser avec un code effectuant du retour arrière. On part du programme calculant la solution aux problèmes des N reines et on en fait plusieurs variantes.

Remarque contrairement à ce qui a été annoncé en cours, on n'aborde pas dans le TP le problème « le compte est bon » (car il y a suffisamment à faire). Ce dernier sera abordé lors du cours 12 de révisions/préparation à l'examen.

1 N-reines état persistant

Lire le fichier `nqueens.ml`. Ce dernier contient le code de la fonction permettant de rechercher toutes les solutions au problème des N-reines. Le but de cette partie est de donner une version alternative du code n'utilisant pas un tableau de booléens mutable, mais gardant simplement la liste des reines déjà posées.

1. Compléter la fonction `compatible : int -> int -> int -> int -> bool`, telle que si on appelle `compatible r1 c1 r2 c2` la fonction renvoie `true` si et seulement si les reines en position `(r1, c1)` et `(r2, c2)` ne sont pas en prise. Pour tester si elles sont sur la même diagonale, il suffit de remarquer pour que si on fait la différence composante par composante des deux coordonnées on obtient une nouvelle coordonnée de la forme (i, i) ou $(i, -i)$

   Indication plutôt que de faire des boucles, on peut raisonner sur les valeurs des coordonnées. En effet il est facile de tester si `(r1, c1)` et `(r2, c2)` sont sur la même ligne ou sur la même colonne.
2. Compléter la fonction `compatible_list : int -> int -> (int * int) list -> bool` qui, si on appelle `compatible_list r c l` vérifie que `(r, c)` est compatible avec toutes les coordonnées de `l`.
3. Copier le code de la fonction `nqueens` en une nouvelle fonction `nqueens_list`. Modifier la fonction `search` pour prendre comme argument supplémentaire la liste des des reines déjà posées. On appellera la fonction `f` sur cette liste, qui représente la solution au problème (lorsque l'on est arrivé au cas de base).
4. Modifier la fonction `main` pour appeler dans le corps de la boucle `for` votre fonction `nqueens_list` et contrôler que vous obtenez les mêmes résultats et un temps d'exécution similaire (le temps peut être meilleur ou un peu moins bon en fonction de la façon dont `compatible_list` est implémentée).

   Attention si votre programme passe du temps à calculer et effectue peu d'affichages, il se peut que ces dernieres restent dans le *buffer* de la sortie standard sans apparaître à l'écran. On peut prendre l'habitude de faire suivre le caractère `\n` dans `Printf.printf` par la séquence `%!` qui force l'affichage, par exemple `Printf.printf "Hello\n%!"`.
5. Lire le code se trouvant dans `utils.ml`. S'en inspirer pour écrire une fonction

   `print_list : (int * int) list -> int -> unit`

   qui affiche l'échiquier de taille `n` et les reines aux positions données.
6. Écrire une fonction de test et l'appeler à la place de `main` pour le problème des 4 reines et constater qu'on a bien les 2 solutions vues en cours.

2 Trouver une solution, rapidement

Dans cette partie on essaye de trouver une solution quelconque sans les énumérer toutes.

1. Écrire une fonction `search_one_solution : unit -> unit` qui similaire à la fonction `main` et qui mesure le temps mis pour trouver une solution pour le problème des N-reines pour N entre 0 et 100. À partir de quelle valeur de N le temps devient il déraisonnable (i.e. supérieur à une dizaine de secondes) ? Vous pouvez interrompre le programme avec `CTRL-C`.

   Indication attention, pour cette fonction on n'a pas besoin de réécrire la fonction `nqueens_list`, il suffit, comme vu en cours, de définir une exception avec `exception Found of (int * int) list` et de faire en sorte que la fonction `f` lance cette exception lorsqu'elle est appelée.

Une meilleure méthode pour trouver une solution parmi les solutions possibles consiste simplement à essayer de trouver un chemin aléatoire vers la solution dans l'arbre de recherche. Les questions qui suivent vont permettre d'écrire une telle recherche aléatoire (dans laquelle on ne fait plus de retour arrière).

3. Copier la fonction `nqueens_list` en une nouvelle fonction `nqueens_random`. Cette dernière ne prend plus de fonction `f` en argument car elle va renvoyer une unique solution si elle la trouve.
   (a) Dans le cas de base de la fonction `search`, renvoyer `Some other_queens` pour signifier que l'on a trouvé une solution
   (b) Dans le cas récursif au lieu d'essayer tour à tour tous les `(row, k)` valide, les générer tous. Si aucun ne peut être généré (on ne peut placer aucune reine) alors renvoyer `None` et s'arrêter, on n'a pas réussi à trouver de solution. Si on a trouvé des positions valides, en choisir une au hasard et continuer la recherche sur celle-ci.

   Indication pour modifier simplement la fonction on peut faire la chose suivante :
   — avant la boucle `for`, créer deux références, l'une vers un entier `c` l'autre vers une liste vide `l`
   — pour chaque nouvelle position valide, incrémenter `c` et ajouter la position à `l`
   — après la boucle, si l'entier vaut `0`, renvoyer `None`, sinon choisir un entier inférieur à c! avec `Random.int` et prendre le couple se trouvant à cette position dans la liste (avec `List.nth`).
4. Écrire une fonction `search_random_solution : int -> int * (int * int) list` qui effectue lance la recherche aléatoire jusqu'à ce que cette dernière renvoie une solution et compte le nombre d'essais. Attention, on prendra soin de traiter les cas de `2` et `3` à part, car ce sont les deux instances du problèmes qui n'ont pas de solution, la recherche aléatoire va donc boucler infiniment.
5. Écrire une fonction de test et constater qu'on peut maintenant trouver une solution en un temps raisonnable (moins de dix secondes) pour N allant jusqu'à 100.