clement.rouvroy@ens.psl.eu

Les n-reines et au-delà

Code *n*-reines, cours de J-C Filliâtre

```c
int t(int a, int b, int c) {
  int f = 1;
  if (a) {
    int d, e = a & ~b & ~c;
    f = 0;
    while (d = e & -e) {
      f += t(a-d, (b+d)*2, (c+d)/2);
      e -= d;
    }
  }
  return f;
}
int main() {
  int n;
  scanf("%d", &n);
  printf("q(%d) = %d\n", n, t(~(~0<<n), 0, 0));
}
```

Type CCINP

Consignes Le candidat respectera le langage imposé (C) et ne devra pas utiliser de librairie hors-programme. Il est invité à **faire des schémas clairs et précis** de ses algorithmes lors des appels au correcteur, c'est à la fois un gain de temps pour les deux et une manière de montrer qu'il a compris ce qu'il manipule. Les preuves écrites doivent être formelles, sauf si la consigne précise que ce n'est pas nécessaire, la rigueur sera évaluée. L'examinateur ne débugera pas votre code, en revanche en cas de doute ("ai-je le droit à telle ou telle librairie ?", "je suis sortie de la VM, comment y revenir ?", "ai-je le droit à une indication pour cette question ?") n'hésitez pas à poser votre question. Elle ne vous dévalorisera pas, si la réponse peut vous retirer des points, l'examinateur vous demandera avant de vous donner la réponse si vous l'acceptez (*si vous n'avez pas de chance, le jour de l'oral il vous retirera les points rien que pour avoir posé la question, par exemple si vous demandez "comment trier une liste en $O(n\,log(n))$?" ou un autre résultat classique du programme, ça sera sûrement retenu contre vous*). Enfin, si l'examinateur n'est pas à votre portée quand vous avez besoin d'une aide / d'une question oral à donner, gardez le bras levé et lisez la suite, ne restez jamais passif.

Le barème n'est la qu'à titre indicatif, votre note finale n'est pas juste `points/total_possible`, il y a un redressement & des ajustements en cas de sujet trop long / trop difficile, donc pas de panique.

Introduction

Les n-reines est un probléme classique de retour sur trace en informatique, le but est de prendre un terrain d'échec de dimension $n \times n$ et de placer n reines dessus sans que deux reines ne puissent s'attaquer. Le sujet consiste à implémenter des algorithmes (proches mais différents) pour résoudre le problème et à voir lesquels sont efficaces.

clement.rouvroy@ens.psl.eu

I - Pendant que ça charge

Question 1 à l'écrit

1. Quelle est la complexité des opératiosn de recherche et de suppression dans un arbre binaire de recherche ?
2. En toute généralité, peut-on affirmer que la hauteur d'un arbre binaire est $O(n)$? $O(\log n)$?
3. Quelle(s) syntaxe(s) permet(tent) de déclarer un type arbre n-aire en OCaml ? Pour ceux qui ne le permettent pas, expliquer les erreurs. On supposera que `etiq_t` est un type correct.

```
type N_arbre1 = Vide | Feuille of etiq_t | Noeud of etiq_t * (N_arbre1 array)
type n_arbre2 = Vide | Feuille of etiq_t | Noeud of etiq_t * array
type n_arbre3 = Vide | Feuille of etiq_t | Noeud of (etiq_t, n_arbre3 array)
type n_arbre4 = Vide | Feuille of etiq_t | Noeud (etiq_t * n_arbre4 array)
type n_arbre5 = Vide | Feuille of etiq_t | Noeud of etiq_t * n_arbre5 array
```

II - Le problème

On se donne un **échiquier**, à savoir une matrice de taille $n \times n$ que l'on créé avec une fonction de type `echiquer creer_terrain(int taille)`. La matrice contient des booléens (on n'utilsera pas des entiers comme booléens, le programme l'interdit), si la case `e[i][j]` est à `true`, alors la case `(i,j)` contient une dame. On veut placer `n` reines sur l'échiquier de sorte qu'aucune d'entre-elles ne puissent se toucher. Pour rappel une reine peut se déplacer en diagonale (dans les 4 diagonales) ou de manière latérale (droite/gauche et haut/bas) comme sur cette image:

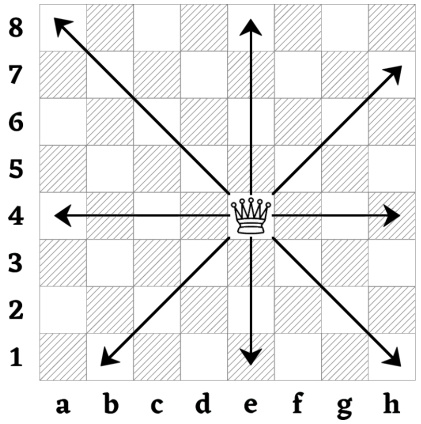

Figure 1: Exemple (Mouvement d'une reine)

Question 2 à l'écrit

1. Peut-on avoir deux dames sur une même ligne ?
2. Peut-on avoir deux dames sur une même colonne ?

III - Les n-reines par recherche exhaustive

Le problème ne semble pas si méchant comme ça: on a un terrain de taille 8×8 au début, et même si on monte à 20×20, le terrain reste assez petit, on peut être tenté de se dire que le bruteforce est une bonne idée, voyons ce qu'il en est.

Question 3 code

Implémenter le type `echiquier` ainsi que la fonction `creer_terrain`, par défaut le terrain ne possède aucune reine. Le type `echiquier` a un deuxième champ pour stocker le n du terrain. (utiliser un `struct`).

Votre type `echiquier` est-il une référence de `struct echiquier` ? Si oui, pourquoi ? Si non, pourquoi ?
Remarque: On ne touchera jamais à la valeur `n` d'un échiquier, une fois set, elle restera constante.

Question 4 code

Ecrire une fonction `bool collision(echiquier e, int i, int j)` qui renvoie vrai si l'ajout d'une reine en case (i, j) créé une attaque et faux sinon. Remarque: Je vous invite à la découper en plusieurs fonctions qui vérifient les différents types de mouvements. Ne cherchez pas à optimiser cette fonction, ce sera l'objet de la dernière sous-partie

Question 5 à l'écrit

Dans la question précédente, on passe i et j dans la fonction pour ne vérifier que la reine (i, j), est-ce vraiment utile ? Gagne-t-on en complexité temporelle dans le pire cas par rapport à une fonction de type `bool collision(echiquier e)` qui vérifierait l'intégralité à chaque fois (que vous n'avez pas à coder). JUSTIFIER SVP.

III.1 - Bruteforce

On va vouloir placer une dame par ligne. D'après la question 2 (oui la réponse se trouve ici), vous ne pouvez avoir qu'une seule dame par colonne. On va utiliser ce principe pour tenter une approche par force brute:

Question 6 code

1. Ecrire un algorithme `?? permutations(int n)` qui génère les permutations de $[\![1; n]\!]$, vous préciserez son type de retour dans le code, pas besoin de l'écrire sur votre copie.
2. Ecrire une fonction `int* bruteforce(int n)` qui renvoie une solution au problème des n-reines par bruteforce en utilisant votre fonction `permutations`.
3. (à l'écrit) Pour quelle valeur de n le temps d'attente dépasse 10 secondes ? Est-ce logique (rapidement) ?

III.2 - Retour sur trace

Une mesure permettant de voir l'efficacité du retour sur trace est **l'arbre de retour sur trace** (terme pas au programme je crois), qui consiste à représenter l'évolution de nos branches dans un arbre n-aire en coupant les branches quand on a une solution partielle qui est fausse. Par exemple,voici l'arbre de retour sur trace pour les n-reines avec $n = 4$:

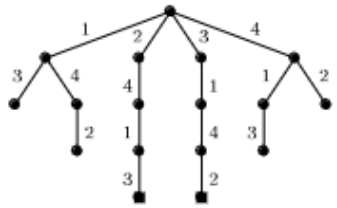

Figure 2: Exemple (Source: The Art Of Computer Programming 4B)

Question 7 à l'écrit

En utilisant vos réponses à la question 2, proposer un algorithme par retour sur trace (terme que vous devez définir à l'écrit au préalable) pour résoudre le problème. Appelez-moi pour valider votre réponse avant de passer à la suite (un seul passage par élève, si je viens et que c'est faux, je vous corrige en direct mais vous n'avez pas les points, appelez-moi quand vous êtes sûrs)

Question 8 code

Implémenter l'algorithme donné à la question précédente. Lancez-le pour des valeurs de n entre 0 et 1000000 (faîtes-le de 1 en 1, ie 0,1,2,3,... svp).

Pour votre information, votre stratégie de retour sur trace a l'arbre de retour sur trace suivant pour $n = 8$ (normalement):

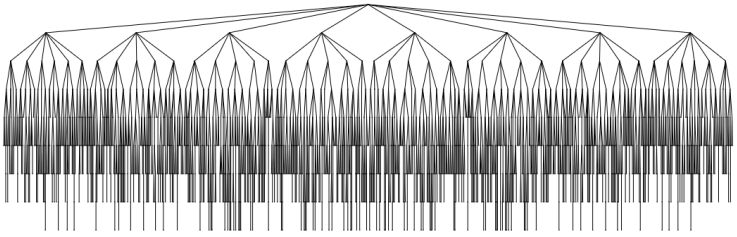

Figure 3: Source: The Art Of Computer Programming 4B

Question 9 à l'écrit

Voici quelques informations sur l'arbre ci-dessus:
- Il a 2057 noeuds
- Son profil est $(1, 8, 42, 140, 344, 568, 312, 92)$ (p_i est le nombre de noeud à la hauteur i)

Combien y a-t'-il de solution au problème des 8-reines ? Quelle est l'amélioration (en %) du nombre de cas visités par rapport à votre bruteforce ?

Question 10 à l'écrit

1. A partir de quelle valeur de n le temps d'attente dépasse les 10 secondes ? Expliquer ce résultat par une étude de la complexité temporelle.
2. Quelle est la complexité en mémoire ?

III.3 - Améliorations

On peut définir le retour sur trace de cette manière:

Algorithm B (*Basic backtrack*). Given domains D_k and properties P_l as above, this algorithm visits all sequences $x_1 x_2 \dots x_n$ that satisfy $P_n(x_1, x_2, \dots, x_n)$.

B1. [Initialize.] Set $l \leftarrow 1$, and initialize the data structures needed later.

B2. [Enter level l.] (Now $P_{l-1}(x_1, \dots, x_{l-1})$ holds.) If $l > n$, visit $x_1 x_2 \dots x_n$ and go to B5. Otherwise set $x_l \leftarrow \min D_l$, the smallest element of D_l.

B3. [Try x_l.] If $P_l(x_1, \dots, x_l)$ holds, update the data structures to facilitate testing P_{l+1}, set $l \leftarrow l + 1$, and go to B2.

B4. [Try again.] If $x_l \neq \max D_l$, set x_l to the next larger element of D_l and return to B3.

B5. [Backtrack.] Set $l \leftarrow l-1$. If $l > 0$, downdate the data structures by undoing the changes recently made in step B3, and return to B4. (Otherwise stop.) ∎

Figure 4: Algorithme Backtracking (Source: The Art Of Computer Programming 4B)

On remarque que la seule étape améliorable (si on ne fait pas d'optimisation tierce) est la vérification de "$P_{l(x_1,\dots,x_l)}$ est vraie", ce qui revient dans notre cas à vérifier qu'il n'y a pas de collision. Cherchons à optimiser un peu cette étape.

Question 11 à l'écrit

Justifier qu'il y a collision en ajoutant la dame numéro j ssi $\exists i < j, x_j - x_i \in \{j - i, 0, i - j\}$.

Au regard de cette équivalence, on propose (c'est une idée très jolie) de ne stocker en mémoire que trois tableaux **de booléens** $\mathcal{A}$ de taille n, $\mathcal{B}$ de taille $2n - 1$ et $\mathcal{C}$ de taille $2n - 1$ tels que:

- $\mathcal{A}[k]$ est vrai ssi $\exists k, x_k = j$
- $\mathcal{B}[j]$ est vrai ssi $\exists k, x_k + k = j$
- $\mathcal{C}[j]$ est vrai ssi $\exists k, x_k + n - k = j$.

Question 12 à l'écrit

Supposons qu'au tour j vous avez accès aux trois tableaux à jour, comment s'en servir pour savoir si il y a une collision ?

Question 13 code

En déduire une version améliorée de votre algorithme de retour sur trace des n-reines.

Heureusement pour vous, vous n'avez pas à calculer l'efficacité de votre algorithme, Donald E. Knuth l'a fait pour vous. Votre première version, pour $n = 16$, utilise en moyenne 98 accès mémoire par noeud de votre arbre de retour sur trace, la deuxième version en moyenne 30 (on passe de 112 milliards à 34 milliards d'appels).

Désolé, ça va être frustrant pour vous mais on ne verra pas plus efficace pour les n-reines, la raison est simple: Les n-reines sont NP-Complets (vous comprendrez en deuxième année), par contre il existe un très bon algorithme (et c'est un algorithme naïf !) aléatoire que vous verrez en deuxième année qui résoud facilement pour $n = 100$ (en moins d'une seconde), mais c'est hors de portée en sup.

IV - Optimisation de pièces d'échecs

Cette partie du TP est plus libre, à vous de développer vos algorithmes de retour sur trace pour résoudre ces questions (interdit de répondre en réfléchissant mathématiquement aux questions, faîtes des algorithmes de retour sur trace)! Vous pouvez les traiter dans l'ordre que vous voulez, l'important c'est de vous faire plaisir, ces problèmes sont sympathiques (vous avez déjà largment la moyenne si vous êtes ici de toute façon)!

Question 14 code

Combien de fous peut-on disposer sur un échiquier 8×8 sans qu'aucun ne se touche ? Remarque: Le fou ne peut que se déplacer en diagonale

Question 15 code

Problème du cavalier

Développer un algorithme (retour sur trace) qui donne une suite de coups à faire sur un cavalier pour qu'il visite chaque case **exactement une fois** sur un échiquier. Le cavalier part d'en haut à gauche.