Rapport projet semestriel

Intelligence Artificielle

Université d'Aix-Marseille
L3 informatique

Travail réalisé par Chloé Bensoussan

Sommaire

Introduction

L'intelligence artificielle (IA) est un domaine très vaste qui touche de nos jours, tous les secteurs. Elle évolue tous les jours et a pour but de posséder l'intelligence humaine et peut-être même la dépasser.

Dans cet UE, nous avons appris que l'IA met en pratique des algorithmes de stratégies, de logiques et de contraintes. C'est ainsi, nous somme parvenus à programmer deux algorithmes célèbres pour résoudre des problèmes à contraintes : le Forward Checking et la Back-Jumping.

Pour tester les algorithmes, nous avons choisi de résoudre 2 problèmes majeurs en informatique : le problème des Dames et le problème des Pigeons.

Le problème des Dames (souvent dite le problème des 8-dames) est un problème classique dans le monde informatique. Son but est de placer toutes les dames dans un échiquier de 8x8 sans avoir 2 dames sur la même ligne, la même colonne et les deux diagonales.
Ce problème simple mais non évidente, est très souvent utilisé pour implémenter des algorithmes.

Le problème des Pigeons consiste à placer n pigeons sur (n-1) nids avec au moins un pigeon par nid puis tous les pigeons ayant des nids différents. Ce problème n'ayant pas de solution, permet de vérifier l'efficacité des algorithmes.

Nous allons voir par la suite les implémentations de la structure, et des algorithmes. Puis nous continuerons sur les problèmes rencontrés lors de la programmation ainsi que les limites du programme. En poursuivant la lecture nous verrons le manuel d'utilisation avant de terminer par le résultat et la comparaison des deux algorithmes implémentés.

Implémentation

I. Structure utilisée

Pour résoudre les deux problèmes énoncés, nous devions implémenter la structure d'un CSP.

```
#define MAX_RELATION   100
#define MAX_VARIABLES  100
#define MAX_DOMAINES   100


typedef struct {
     int relation[MAX_RELATIONS][MAX_RELATION];
}Relations;


typedef struct {
     int nb_variables;
     int nb_valeurs;
     int domaines[MAX_VARIABLES][MAX_DOMAINES];
     Relation *contraintes[MAX_VARIABLES][MAX_VARIABLES];
}Csp;
```

Tout d'abord je défini 3 constantes permettant de limiter les tailles des tableaux. La structure `Csp` contient le nombre de variables et le nombre de valeurs qui représente le nombre de domaines possibles au total.

La matrice `domaines[][]` est affecté à 1, si la variable `i` peut prendre la valeur `j` avec `domaines[i][j]`.

Le tableau `contrainte[][]` est un pointeur pointant sur une matrice de type `Relations` s'il existe une contrainte entre deux variables `i` et `j` avec `contraintes[i][j]`. Sinon, elle prend la valeur `NULL`.
Lorsqu'il existe une contrainte entre 2 variables, on remplit la matrice `relation[][]`.

Pour une valeur k de la première variable, et une valeur `l` pour la deuxième variable, on affecte à 1 la matrice si les deux variables peuvent prendre respectivement les valeurs i et j avec `contrainte[i][j]->relation[k][l]`. Sinon, on affecte à -1.

II. Forward Checking

L'algorithme Forward Checking, est une méthode complète, qui filtre les domaines des variables non instanciées pour trouver une solution rapidement.

Voici l'algorithme que j'ai suivi pour programmer :

```
tant que toutes les variables ne sont pas instanciées, faire :

    sélectionner le premier domaine disponible de la variable courante;

    s'il n'y a plus de domaine disponible ou ne vérifie pas une
    contrainte, faire :
            Backtracker de un niveau (revenir à la variable précédente);

    sinon :
            filtrage des domaines (en colonne et en diagonales);
    fin si;

fin tant que
```

III. Filtrages

Chaque problème possède son propre filtrage. C'est pourquoi j'ai implémenter deux fonctions de filtrage : filtrage des Pigeons et le filtrages des Dames.

Le filtrage des Pigeons est très simple : il suffit de supprimer les domaines déjà utilisés dans toutes les variables.

Néanmoins pour celui des Dames, il se doit d'enlever les domaines utilisées dans toutes les variables, ainsi que les domaines se trouvant dans les diagonales des domaines utilisés.

IV. Backjumping

L'algorithme Backjumping possède un algorithme ressemblant à celui de Forward Checking.

A chaque vérification de contraintes, si la variable courante viole une contrainte, on change son domaine. Si il a testé tous ses domaines disponibles, on retourne à la variable qui cause le viole. Dans ce cas, l'algorithme remets les domaines filtrés avec la variable courante à libre.

```
tant que toutes les variables ne sont pas instanciées, faire :

    sélectionner le premier domaine disponible de la variable courante;

    s'il n'y a plus de domaine disponible, faire :
            Backtracker de un niveau (revenir à la variable précédente);

    sinon :
            tester vérification des contraintes;
            si une contraintes est violée, faire :
                    Backtracker de un niveau (revenir à la variable
                        précédente);
                    remettre les domaines supprimées par la variable
                        courante;
                    supprimer le domaine courant pour la variable
                        courant;
            sinon :
                    affectation de domaine pour la variable;
                    passer à la variable suivante;

fin tant que
```

Les problèmes rencontrés

J'ai rencontré plusieurs problèmes lors de la programmation de ce projet.

La première a été la structure de la matrice `relation[][]` pointée par une matrice `contraintes[][]`. Après plusieurs essai, j'ai décidé de créer une structure `Relation`, qui permet de mieux manipuler les ces deux matrices.

De plus, pendant toute la période de programmation de ce projet, j'avais un problème d'initialisation de cette table. En effet, les relations ne voulant pas bien s'implémenter (il y avait des 1 alors que la relation n'était pas possible), j'ai passé beaucoup de semaine sur ce problème ce qui m'a ralentit dans la code des algorithmes par la suite. Ce problème est aujourd'hui résolu grâce à l'allocation dynamique de la matrice avant de définir les valeurs dans celle-ci.

Travail inachevé et limites du programme

Aujourd'hui, le projet n'est pas terminé. Par manque de temps et ayant eu des problèmes pendant longtemps, je n'ai pas pu terminer l'implémentation des fonctions heuristiques.

De plus, ce projet comprend une limite : ayant une capacité de stockage de pile limitée à 1000, le programme n'affichera que les 1000 premières solutions lors de la recherche de toutes les solutions. C'est à dire, à partir de 11 variables, le programme ne pourra élaborer l'arbre complet.

Manuel d'utilisation

Pour débuter la démonstration, déplacer le répertoire dans un endroit. Puis à partir d'un terminal, accéder au répertoire.
Taper `make all` dans la ligne de commande comme si dessous.

```
[sadaakira-sama:IA-master ChloeYukino$ make all
gcc -g -Wall -w -ggdb -c generateurPigeon.c
gcc -g -Wall -w -ggdb -c generateurDame.c
gcc -g -Wall -w -ggdb -c util.c
gcc -g -Wall -w -ggdb -c pile.c
gcc -g -Wall -w -ggdb -o csp csp.c generateurPigeon.o generateurDame.o util.o pile.o
sadaakira-sama:IA-master ChloeYukino$
```

Si aucun message d'erreur ne s'affiche, nous pouvons procéder à la suite.

Lancer la commande `./csp` permettant de lancer le programme.
Une liste de question s'affichera au fur et à mesure pour définir le problème à résoudre (voir l'image ci-dessous).

```
[sadaakira-sama:IA-master ChloeYukino$ ./csp
Quel problème résoudre ?
        0-le problème des Dames
        1-le problème des Pigeons
0
Quel algorithme utiliser ?
        0-Forward Checking
        1-Backjumping
0
Travailler avec combien de variable ?   4
Voulez-vous rechercher une ou toutes les solution ?
        0-une solution
        1-toutes les solutions
0
```

Par exemple, si vous voulez résoudre le problème des Dames, en Back Jumping avec 8 variables, il vous faudra enchaîner les valeurs 0, 1, 8, 0.

Une fois toutes les valeurs insérées, le programme effectue l'algorithme demandé, puis affiche le résultat : il nous affiche d'abord si une (ou plusieurs) solution(s) existe(nt) avec son temps d'exécution.

```
----------------------------------------
Il y a une solution.
Temps d'éxectution de l'algorithme : 0.005833 ms
----------------------------------------

Afficher le(s) solution(s)?
        0-oui
        1-non
```

Nous avons désormais, la possibilité d'affiche les solutions trouvées ou pas.
Voici un exemple d'affichage de solution finale :

```
Afficher le(s) solution(s)?
        0-oui
        1-non
0
***
[0 1]
[1 3]
[2 0]
[3 2]
```

Le programme est maintenant terminé. Pour recommencer le lancement du programme, recommencez les étapes d'opérations à partir du la ligne de commande `./csp`.

Comparaison des deux algorithmes

Voici un tableau représentant les temps d'exécution du programme en fonction de algorithme (Forward Checking et Back Jumping), ainsi que du nombre de variables.

nombre de variables	**Forward Checking** (en ms)	**BackJumping** (en ms)
10	0.021	0.038
11	0.014	0.028
12	0.053	0.104
13	0.028	0.055
14	0.4	0.944
15	0.3	0.743
16	2.073	5.688
17	1.459	3.250
18	9.808	26.645
19	0.686	1.790
20	46.112	156.101
30	21057.33	89719.02

Nous pouvons constater que les résultats sont très proches aux premières lignes (10 à 13 variables) . Cependant, plus le nombre de variables augmente, plus l'écart entre les deux algorithmes augmente. Ainsi, pour 20 et 30 variables, un grand écart qui permet d'affirmer que l'algorithme Forward Checking est plus efficace que Back Jumping en termes de temps.