[C] french

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene
Faculté d'électronique et d'informatique
Département d'informatique


# Rapport de projet

Module : **Programmation par contrainte**

Master 1 SII

## Implémentation de l'algorithme Look-Ahead avec filtrage PC1 et PC2

- Réalisé par :

  **BENHADDAD Wissam**

  **BOURAHLA Yasser**

**5-12-2018**

# Table des matières

french



# Table des figures

french

# Liste des tableaux

french

CHAPITRE 1

# INTRODUCTION

## Problématique et objectifs

Dans le domaine de la résolution de problèmes, nous faisons souvent appel à des techniques et algorithmes dits **intelligents** pour la recherche d'éventuelles solutions. Les problèmes de satisfaction de contraintes en sont un exemple, ainsi le paradigme de la programmation par contraintes à été introduit pour modéliser ces et résoudre ces types de problèmes, en introduisant des méthodes venant de la théorie des ensembles et de la logique mathématique.

Dans ce projet, nous allons proposer notre propre implémentation d'une version d'un algorithme bien connu de résolution d'un PSC[1] , à savoir l'algorithme **Look-Ahead** avec filtrage **PC1** et **PC2**.

Avant de commencer nous devons tout d'abord définir dans ce chapitre quelques notions théoriques(section suivante), puis présenter le squelette des algorithmes utilisés, puis nous définirons dans le chapitre suivant les principales composantes de notre système, nous finirons ensuite par une comparaison entre les deux types de filtrage utilisés sur deux PSC, l'un connu et un autre dont les paramètres sont générés aléatoirement. Nous conclurons ce travail par une conclusion générale récapitulant l'ensemble des connaissances apprises.

## Définitions

Dans cette section nous allons définir les composantes théoriques d'un PSC, ainsi que ceux d'un algorithme de résolution d'un PSC.

### Problème de Satisfaction de Contraintes

Un PSC, est formellement définie comme étant un triplet $< X, D, C >$, où :

1. Problème de Satisfaction de Contraintes

- $X = \{x_1, x_2, ..., x_n\}$ est l'ensemble des variables du PSC.
- $D = \{D_1, D_2, ..., D_n\}$ est l'ensemble des domaines des variables, c'est-à-dire que pour tout $k \in [1; n]$ nous avons $x_k \in D_k$.
- $C = \{C_1, C_2, ..., C_m\}$ est un ensemble de contraintes. Une contrainte $C_i = (X_i, R_i)$ est définie par l'ensemble $X = \{x_{i_1}, x_{i_2}, ..., x_{i_n}\}$ des variables sur lequel elle porte et la relation $R \subset D_{i_1} \times ... \times D_{i_k}$ qui définit l'ensemble des valeurs que peuvent prendre simultanément les variables de $X_i$

## Consistance d'un PSC

Un PSC est dit consistant si l'on parvient à trouver une instanciation des variables $X_i$ tel que toutes les contraintes $C_j$ soient satisfaites en même temps.

Pour ce faire nous utiliserons des algorithmes de recherche de solutions dit intelligents, parmi eux l'algorithme **Look-Ahead**, l'idée est d'instancier des variables une à une en essayant d'appliquer un filtrage des domaines des variables non encore instanciées, ainsi, ç l'instanciation d'une variable, le choix de sa valeur sera idéalement plus restreint, et de ce fait le temps de recherche et de détection d'inconsistance sera plus court.

Look-ahead applique au niveau de chaque noeud un filtrage de consistance de chemin [2]. Nous allons voir deux algorithme de consistance de chemin, PC1 et PC2.

**Remarque** : Pour les détails théorique, il faut se référer au support de cours qui donne de très bonnes explications. Dans ce rapport nous ne présenterons que les algorithmes des différentes méthodes

**PC1** Nous devons d'abord définir une fonction **REVISE_PC** :

[H] InputEntréeOutputSortie genGenererCandidats

$(X_i, X_k, X_j), (X, D, C)$ $VRAI/FAUX$

$temp \leftarrow M_P[i,j] \bigcap M_P[i,k]^{\circ} MP[k,k]^{\circ} MP[k,j]$ $temp \neq M_P[i,j]$ $M_P[i,j] \leftarrow temp$ $VRAI$

$FAUX$

L'algorithme PC1 est le suivant :

[H] InputEntréeOutputSortie genGenererCandidats revREVISE_PC

$(X, D, C)$ $(X, D, C)$

$Q \leftarrow (X_i, X_k, X_j) \in X^3 : \neg(X_i = X_k = X_j)$ $R = FAUX$ $R \leftarrow FAUX$ $(X_i, X_k, X_j) \in Q$ $(X_i, X_k, X_j), (X, D, C) = VRAI$ $R \leftarrow VRAI$ $(X, D, C)$

L'algorithme PC2 est le suivant :

[H] InputEntréeOutputSortie genGenererCandidats revREVISE_PC

2. `https://en.wikipedia.org/wiki/Local_consistency`

$(X, D, C)$ $(X, D, C)$

$Q \leftarrow (X_i, X_j) \in X^2 : (i \leq j)$ et il existe une contrainte entre $X_i$ et $X_j$

$Q \neq \emptyset$ Prendre une paire $(X_i, X_j)$ de variables de $Q$, et l'en supprimer $: Q \leftarrow Q - \{(X_i, X_j)\}$

$k = 1n \wedge \neg(k = i = j)$ $temp \leftarrow M_P[i,k] \bigcap M_P[i,j]^\circ M_P[j,j]^\circ M_P[j,k]$ $temp \neq M_P[i,k]$ $M_P[i,k] \leftarrow temp$ $M_P[k,i] \leftarrow (temp)^T$ $i \leq k$ $Q \leftarrow Q \bigcup \{(X_i, X_k)\}$ $Q \leftarrow Q \bigcup \{(X_k, X_i)\}$

$temp \leftarrow M_P[k,j] \bigcap M_P[k,i]^\circ M_P[i,i]^\circ M_P[i,j]$ $temp \neq M_P[k,j]$ $M_P[k,j] \leftarrow temp$ $M_P[j,k] \leftarrow (temp)^T$

$k \leq j$ $Q \leftarrow Q \bigcup \{(X_k, X_j)\}$ $Q \leftarrow Q \bigcup \{(X_j, X_k)\}$

$(X, D, C)$

Enfin l'algorithme look-ahead avec un choix de filtrage :
[H] InputEntréeOutputSortie genGenererCandidats pcPC1 pccPC2

$A, (X, D, C)$ $VRAI/FAUX$

$Consistance \leftarrow (X, D, C)$ OU $Consistance \leftarrow (X, D, C)$

$\neg Consistance$ $FAUX$

toutes les variables de $X$ sont instanciées $VRAI$ $Xi \leftarrow$ variable de $X$ qui n'est pas encore instanciée

$V_i \in D(X_i)$ $D\prime(X_i) = \{Vi\}$ $D\prime \leftarrow (D - \{D(X_i)\}) \bigcup D\prime(X_i)$ $Look-Ahead(A \bigcup \{(X_i, V_i)\}, (X, D\prime, C))$ $VRAI$ $FAUX$

# CHAPITRE 2

# SOLUTION PROPOSÉES ET IMPLÉMENTATION

Dans ce chapitre nous allons voir l'implémentation de l'algorithme lookAhead avec filtrage PC2 tel que décrit dans le chapitre précédent.

## Outils utilisés

Pour l'implémentation de LookAhead nous avons opté à utiliser le langage de programmation C++ qui à la fois garde la rapidité du C tout en donnant la possibilité de modélisation orienté objet, et ceci en utilisant l'IDE de JetBrains CLion.



## Implémentation

Notre implémentation de LookAhead avec filtrage PC2 se compose de trois classes principales : Domains, Constraint et Solver.

### Domains

Cette classe permet de gérer les domaines des variables, elle contient pour chaque variable son domaine initiale et son domaine courant, c'est à dire après application des algorithmes de filtrage

ou instanciation de la variable. Les domaines sont représentés sous forme de listes d'entier.

## Constraint

Chaque contrainte C(Xi,Xj) est représentée par une instance de cette classe, cette dernière contient une matrice nxm où n et m sont les tailles des domaines des variables Xi et Xj respectivement. Cette classe facilite les opérations sur les contraintes notamment la transposition, la composition et l'intersection.

```
class Constraint
{
public:
    const static int UNIVERSAL = 1;
    const static int IDENTITY = 0;
    int n,m;
    int** M;

    Constraint(int n, int m, int** R);
    Constraint(int n, int m);
    Constraint(); // constructors
    void setConstraint(int n, int m,int flag); //set constraint as UNIVERSAL or IDENTITY
    bool isZero(); // checks if constraint is all zero
    Constraint& transpose(); // transpose the constraint
    void print(); // prints the contsraint matrix
    int* &operator[](int i); // overload the [] operator to simplify access
    Constraint &operator&(Constraint& con); // overload the & operator to simplify intersection
    Constraint &operator*(Constraint& con); // overload the * operator to simplify the composition
    void operator=(Constraint& con); // overload the = operator to simplify the copy function
    bool operator==(Constraint& con); // overload the == operator to simplify comparison

};
```

FIGURE 2.2 – L'espace de dessin

## Solver

Un CSP est représenté par cette classe. Elle contient donc les paramètres composant un CSP <X,D,C> : les variables, leurs domaines ainsi que les contraintes entre eux.
Nous avons implémenté dans cette classe les algorithmes PC1, PC2 et lookAhead ainsi que la possibilité de choisir la variable à instancier en tirage aléatoire ou en suivant une heuristique.

**LookAhead** L'algorithme lookAhead a été implémenté comme décrit dans la figure fig :LookAhead :

- On applique un filtre et on teste s'il y une inconsistance, sinon si toute les variables ont été instanciées.
- On choisit la prochaine variable à instancier soit suivant une heuristique ou un ordre aléatoire.
- On instancie la variable avec les valeurs possibles dans le domaine (qui a été mis à jour par le filtre).
- On ajoute dans la queue le couple (i,i) pour le prochain appel de PC2.

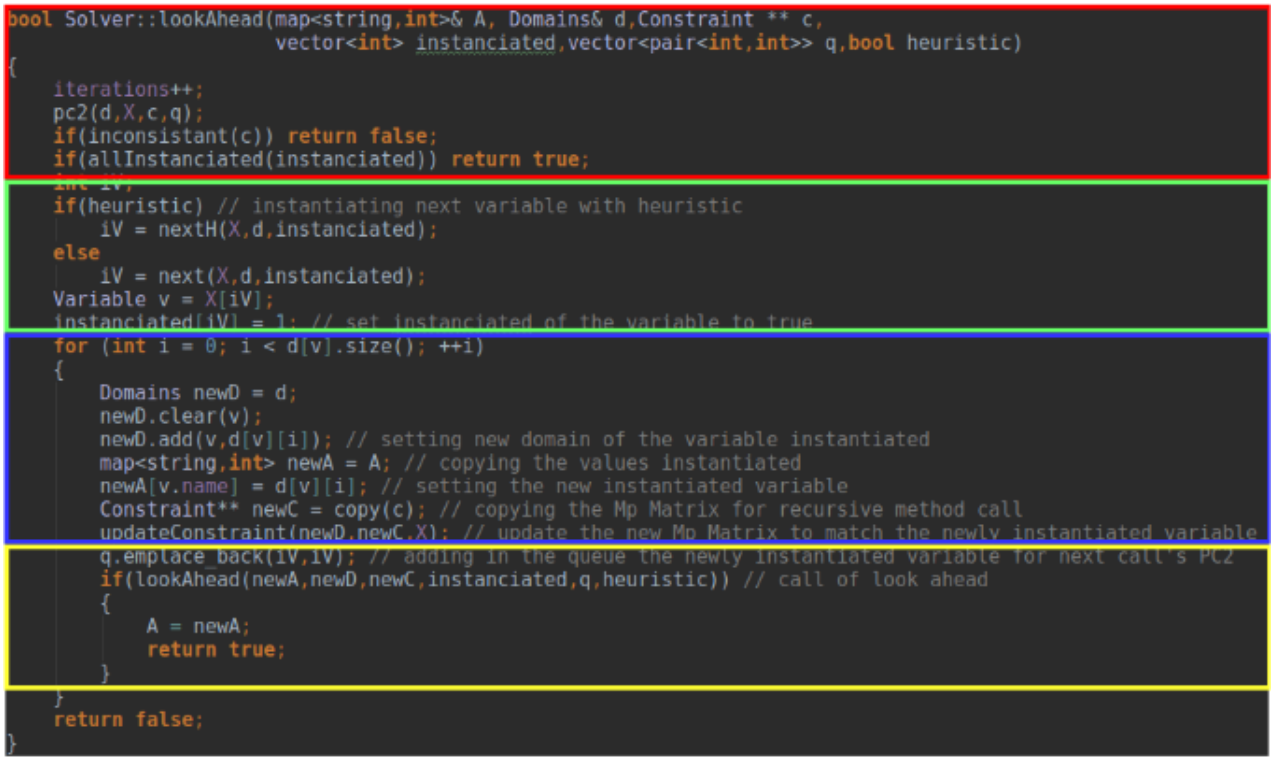


FIGURE 2.3 – L'algorithme lookAhead

**Filtrage** Nous avons implémenté les deux algorithmes de filtrage par chemin PC2 et PC1 :

- PC1 passe par toutes les variables à chaque fois et applique revise, si une entrée de la matrice Mp change, on refait la même chose.

```
void Solver::pc1(Domains& d, vector<Variable>& x, Constraint ** c,vector<int> instanciated)
{
    bool done;
    do
    {
        done = true;
        for (int i = 0; i < X.size(); ++i)
        {
            for (int j = 0; j < X.size(); ++j)
            {
                for (int k = 0; k < X.size(); ++k)
                {
                    if (revise(i, j, k, d, c)) done = false;
                }
            }
        }
    }
    while(!done);
    updateDomain(d,c,x);
}
```

FIGURE 2.4 – L'algorithme PC1

```
bool Solver::revise(int i, int j, int k, Domains d, Constraint **c)
{

    Constraint temp = c[i][j]&(c[i][k]*c[k][k]*c[k][j]);
    if(temp == c[i][j]) return false;
    c[i][j] = temp;
    return true;
}
```

FIGURE 2.5 – La fonction revise de PC1

- PC2 quant à lui utilise une file pour garder les variables dont les entrées de la matrice Mp ont changé afin qu'il les traite.

```
void Solver::pc2(Domains &d, vector<Variable> &x, Constraint **c, vector<pair<int,int>> q)
{
    while (!q.empty())
    {
        pair<int,int> p = q[0];
        q.erase(q.begin());
        int i = p.first,j = p.second;
        for (int k = 0; k < x.size(); ++k)
        {
            if(k == i && k == j)continue;
            Constraint temp = c[i][k]&(c[i][j]*c[j][j]*c[j][k]);

            if(!(temp == c[i][k]))
            {
                c[i][k] = temp;
                c[k][i] = temp.transpose();
                if(i<k) q.emplace_back(i,k); else q.emplace_back(k,i);
            }
            temp = c[k][j]&(c[k][i]*c[i][i]*c[i][j]);
            if(!(temp == c[k][j]))
            {
                c[k][j] = temp;
                c[j][k] = temp.transpose();
                if(k<j) q.emplace_back(k,j); else q.emplace_back(j,k);
            }
        }
    }
    Domains newD = d;
    updateDomain(d,c,x);
}
```

FIGURE 2.6 – L'algorithme PC2

**remarque :** la fonction **updateDomain** mis à jour les domains de variables, dans les structures de listes cités avant dans ce chapitre.

## Génération aléatoire de CSP

Afin de comparé les deux filtres PC1 et PC2 ainsi que le choix de variables aléatoire ou avec heuristique, nous avons implémenté une fonction qui génère un CSP aléatoire selon les paramètres suivant :

- Nombre de variables : c'est le nombre de variables du CSP.

- Taille du domaine tailleD : la taille des domaines des variables varie entre tailleD*0,5 et tailleD*1,5 aléatoirement.
- Difficulté : ce paramètre représente la probabilité d'avoir un 0 dans la matrice d'une contrainte du CSP à générer. Plus ce paramètre est élevé, plus le CSP est difficile à satisfaire.

```
Solver generateCSPRandom(int numVariables,int domainSize, double difficulty)
{
    vector<Variable> X;
    srand(time(NULL));
    // vecteur de variables
    for (int i = 0; i < numVariables; ++i)
        X.emplace_back("g"+std::to_string(i));
    // creation des domaines de variables
    Domains D(X);
    for (int i = 0; i < numVariables; ++i)
    {
        vector<int> domain;
        // taille du domaine aleatoire en suivant le parametre "domainSize"
        int size = random(domainSize/2+1,domainSize*1.5);
        for (int j = 0; j < size; ++j)
            domain.push_back(j+1);
        D.setDomain(X[i],domain);
    }
    // creation de la class Solver avec les parametres variables/domaines
    Solver solver(D,X);
    // remplissage de la matrice Mp de contraintes
    for (int i = 0; i < numVariables; ++i)
        for (int j = i+1; j < numVariables; ++j)
        {
            int N = D[X[i]].size(),M = D[X[j]].size();
            // l'entree i,j de la matrice Mp a la taille N,M les tailles des domaines de variables i,j
            // on genere une contrainte aleatoire en suivant la difficultee donnee en parametre
            solver.setConstraint(i,j,generateRandomConstraint(N,M,difficulty,i==j));
        }
    return solver;
}
```

FIGURE 2.7 – L'algorithme PC2

# Résultats

Nous avons essayé l'algorithme lookAhead d'abord sur un problème connu, le problème des N-Reines. Ensuite nous avons généré des CSP aléatoirement avec des difficultés variables pour tester l'algorithme.

## N-Reines

Pour tester les deux filtres PC1 et PC2 avec et sans heuristiques, nous avons généré le problème de N-Reines pour $N \in [4 - 14]$. les résultats sont résumé dans les figures suivantes :

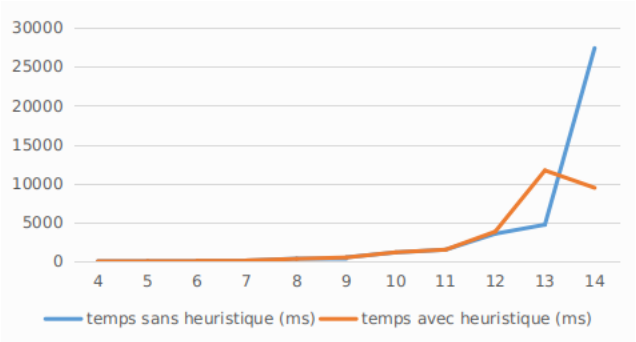


FIGURE 2.8 – Résultats de l'application de PC1 sur le problème des N-Reines

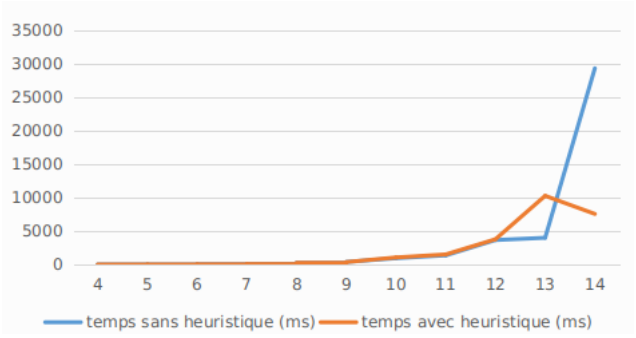


FIGURE 2.9 – Résultats de l'application de PC2 sur le problème des N-Reines

On remarque que dans les deux cas l'heuristique ne donne pas toujours de meilleurs résultats. On remarque aussi que PC2 généralement surpasse PC1 comme le montre les tableaux suivants (les valeurs en rouge sont les valeurs ou PC1 a surpassé PC2) :

| nombre de reines | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| temps sans heuristique (ms) | 2 | 8 | 48 | 85 | 269 | 382 | 937 | 1359 | 3659 | 3999 | 29357 |
| temps avec heuristique (ms) | 3 | 8 | 35 | 71 | 258 | 362 | 1056 | 1507 | 3779 | 10277 | 7567 |

FIGURE 2.10 – temps d'exécution de PC2 par nombre de reines

| nombre de reines | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| temps sans heuristique (ms) | 6 | 18 | 57 | 97 | 341 | 506 | 1143 | 1486 | 3542 | 4703 | 27391 |
| temps avec heuristique (ms) | 5 | 17 | 62 | 112 | 317 | 493 | 1159 | 1492 | 3816 | 11680 | 9446 |

FIGURE 2.11 – temps d'exécution de PC1 par nombre de reines

## Génération aléatoire de CSP

Comme nous l'avons vu dans la partie précédente, l'algorithme de génération aléatoire de CSP dépend de trois valeurs : la difficulté, le nombre de variable et le domaine des variables. Dans cette partie nous allons voir comment la variation de un de ces paramètre affecte elle le temps d'exécution de l'algorithme lookAhead avec filtrage PC2.

**Difficulté** La difficulté représente la probabilité d'avoir un 0, donc si cette probabilité est très faible on arrive à satisfaire le CSP rapidement, même chose si elle est très élevé le filtrage PC2 trouve l'inconsistance rapidement. La figure suivante montre l'exécution de l'algorithme en prenant en paramètre le nombre de variable à 10 et le domaine qui varie de 9 à 27, la difficulté quant à elle est dans une échelle de 0 à 20 où la probabilité d'avoir 0 égale à 1 quand la difficulté est égale à 20.

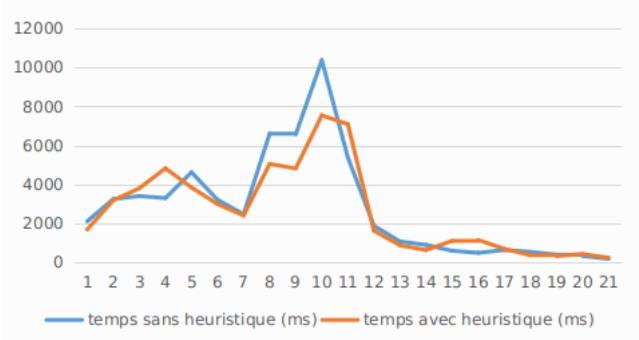


FIGURE 2.12 – Variation du temps d'exécution selon la difficulté

**Nombre de variables** Plus on a un nombre de variable est grand, plus le temps de calcule augmente. Dans la figure suivante la difficulté a été fixée à 0,5 tandis que les tailles des domaines de variables varient entre 8 et 22.

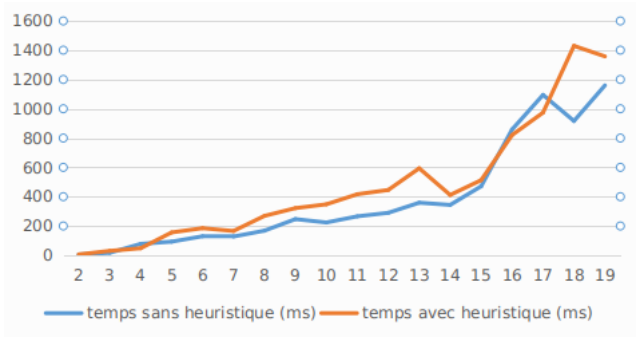


FIGURE 2.13 – Variation du temps d'exécution selon le nombre de variables

**Taille du domaine** La taille du domaine est elle aussi relié proportionnellement au temps d'exécution. Dans la figure suivante la difficulté a été fixée à 0,5 tandis que le nombre de variables varient est à 8.

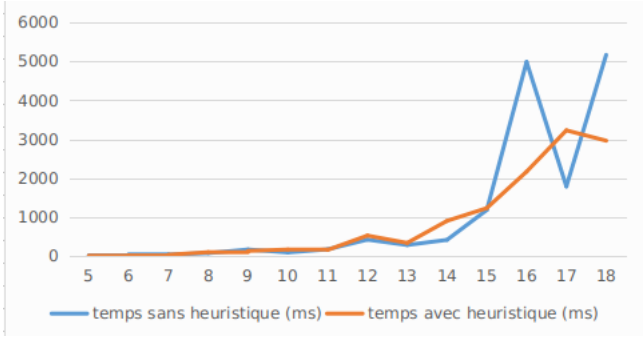


FIGURE 2.14 – Variation du temps d'exécution selon les tailles des domaines

# CHAPITRE 3

# CONCLUSION GÉNÉRALE

Au terme de projet nous pouvons donc tirer les conclusions suivantes :

- Pour ce qui est des petites instances ,l'ajout d'une heuristique(que ce soit avec filtrage PC1 ou PC2) à l'algorithme look-ahead n'influence pas beaucoup sur le temps d'exécution si le nombre de variable est petit, mais plus ce nombre augmente, plus l'apport de l'heuristique se fait ressentir. Cela dépend bien sur du choix de cette dernière.
- Pour un nombre de contraintes et de variables fixe, et une difficulté moyenne(existence de solutions complexes), le filtrage par PC2 et avec l'utilisation d'une heuristique diminue notablement le temps d'exécution.
- avec l'utilisation du filtrage PC2 au lieu de PC1, l'algorithme LOOK-AHEAD rend généralement une réponse plus rapidement.