

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 : **Vision artificielle par ordinateur**

Master 1 SII

**Réalisation d'une application de painting  
en utilisant une détection de couleurs**

- Réalisé par :

**BENHADDAD Wissam**

**BOURAHLA Yasser**

29-11-2018

# Table des matières

<b>Table des matières</b>	<b>2</b>
<b>Table des figures</b>	<b>2</b>
<b>Liste des tableaux</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Problématique et objectifs . . . . .	2
<b>2 Solution proposées et implémentation</b>	<b>3</b>
2.1 Outils utilisés . . . . .	3
2.2 Implémentation . . . . .	3
2.2.1 Domains . . . . .	3
2.2.2 Constraint . . . . .	4
2.2.3 Solver . . . . .	4
2.2.4 Génération aléatoire de CSP . . . . .	6
2.3 Résultats . . . . .	7
2.3.1 N-Reines . . . . .	7
2.3.2 Génération aléatoire de CSP . . . . .	8
<b>3 Conclusion générale</b>	<b>11</b>

# Table des figures

2.2 L'espace de dessin . . . . .	4
2.3 L'algorithme lookAhead . . . . .	5
2.4 L'algorithme PC1 . . . . .	5
2.5 La fonction revise de PC1 . . . . .	6
2.6 L'algorithme PC2 . . . . .	6
2.7 L'algorithme PC2 . . . . .	7
2.8 Résultats de l'application de PC1 sur le problème des N-Reines . . . . .	8
2.9 Résultats de l'application de PC2 sur le problème des N-Reines . . . . .	8
2.10 temps d'exécution de PC2 par nombre de reines . . . . .	8
2.11 temps d'exécution de PC1 par nombre de reines . . . . .	8

2.12	Variation du temps d'exécution selon la difficulté . . . . .	9
2.13	Variation du temps d'exécution selon le nombre de variables . . . . .	9
2.14	Variation du temps d'exécution selon les tailles des domaines . . . . .	10

## Liste des tableaux

## CHAPITRE 1

### INTRODUCTION

#### Problématique et objectifs

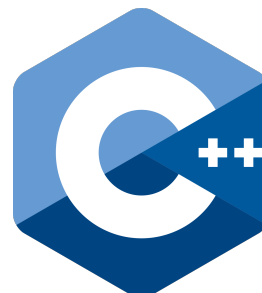
## CHAPITRE 2

# 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 instantiation de la variable. Les domaines sont représentés sous forme de listes d'entier.

## Constraint

Chaque contrainte  $C(X_i, X_j)$  est représentée par une instance de cette classe, cette dernière contient une matrice  $n \times m$  où  $n$  et  $m$  sont les tailles des domaines des variables  $X_i$  et  $X_j$  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 constraint 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  $\langle X, D, C \rangle$  : 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 2.3 :

- On applique un filtre et on teste s'il y a une inconsistance, sinon si toutes 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.

```

bool Solver::lookAhead(map<string,int>& A, Domains& d, Constraint ** c,
                      vector<int> instantiated, vector<pair<int,int>> q, bool heuristic)
{
    iterations++;
    pc2(d,X,c,q);
    if(inconsistent(c)) return false;
    if(allInstantiated(instantiated)) return true;
    int iV;
    if(heuristic) // instantiating next variable with heuristic
        iV = nextH(X,d,instantiated);
    else
        iV = next(X,d,instantiated);
    Variable v = X[iV];
    instantiated[iV] = 1; // set instantiated of the variable to true
    for (int i = 0; i < d[v].size(); ++i)
    {
        Domains newD = d;
        newD.clear(v);
        newD.add(v,d[v][i]); // setting new domain of the variable instantiated
        map<string,int> newA = A; // copying the values instantiated
        newA[v.name] = d[v][i]; // setting the new instantiated variable
        Constraint** newC = copy(c); // copying the Mp Matrix for recursive method call
        updateConstraint(newD,newC,X); // update the new Mp Matrix to match the newly instantiated variable
        q.emplace back(iV,iV); // adding in the queue the newly instantiated variable for next call's PC2
        if(lookAhead(newA,newD,newC,instantiated,q,heuristic)) // call of look ahead
        {
            A = newA;
            return true;
        }
    }
    return false;
}

```

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> instantiated)
{
    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  $M_p$  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 domaines 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("a"+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 :

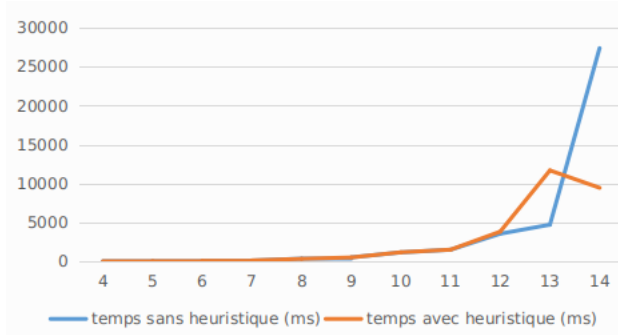


FIGURE 2.8 – Résultats de l'application de PC1 sur le problème des N-Reines

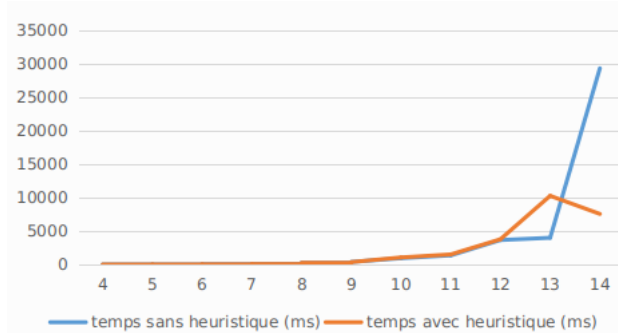


FIGURE 2.9 – Résultats de l'application de PC2 sur le problème des N-Reines

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 où 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 élevée 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.

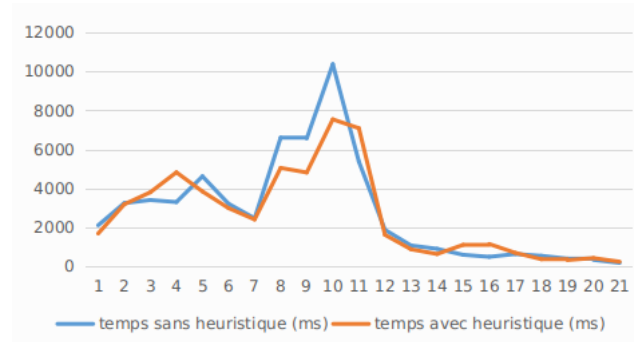


FIGURE 2.12 – Variation du temps d’exécution selon la difficulté

**Nombre de variables** Plus on a un nombre de variable est grand, plus le temps de calcul 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.

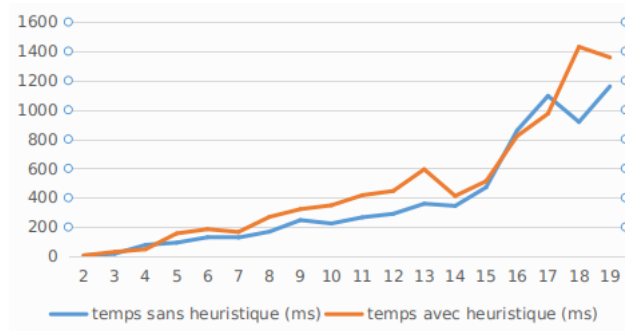


FIGURE 2.13 – Variation du temps d’exécution selon le nombre de variables

**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.

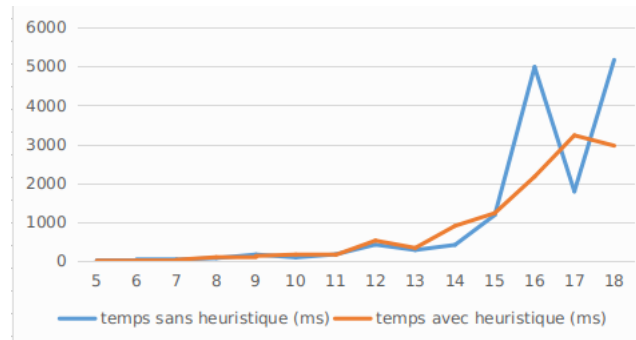


FIGURE 2.14 – Variation du temps d'exécution selon les tailles des domaines

CHAPITRE 3

CONCLUSION GÉNÉRALE