Programmation logique avec contraintes (CLP) Etude et application au puzzle : 'Send plus More equal to Money'

Tayeb LEMLOUMA [†] [*]
Abdelmadjid BOUDINA [†] [*]

[†] *Institut d'informatique, U.S.T.H.B, B.P .32*
El-Alia Bab Ezzouar, Alger 16111, ALGERIE.
Tel / fax 213 2 24 76 07

1 Introduction

La programmation logique (PL) est le fruit des recherches menées par R. Kowalski [Kow 74] et A. Colmerauer [COL 90] sur un sous ensemble de la logique des prédicats du premier ordre à savoir les clauses de Horn.

La programmation logique constitue un paradigme de programmation simple et déclaratif. Ce paradigme reflète une méthodologie de programmation parfaite, le fait qu'il distingue entre l'aspect logique et l'aspect contrôle des programmes. Cette distinction n'est pas appliquée dans les autres styles de programmation.

La principale différence entre les langages de programmation logique, tel que "Prolog", et les langages 'classiques', tel que "Pascal" est que : ces derniers sont de nature impérative, i.e. il faut décrire le problème à résoudre selon un algorithme, alors que les langages logiques, sont de nature déclarative, cela veut dire qu'il suffit d'indiquer au système les données du problème à traiter. Le moteur d'inférence qui permet de mettre en relation ces données, rend alors toutes les solutions. Ainsi avec cette nouvelle vision, on peut résoudre des problèmes complexes sans recourir à des techniques algorithmiques [CON 86].

La souplesse de la PL est assurée grâce aux concepts logiques qui utilisent une manière de spécification des informations décrivant le problème à résoudre en se basant sur la logique du premier ordre. L'aspect contrôle, i.e. la manière dont ces informations doivent être traitées afin de résoudre le problème posé, est assuré par un mécanisme de preuve de théorème appelé "SLD_résolution" (SLD : Selected Lineary Defined Resolution [ROB 65]).

Afin de combler la principale lacune de la PL, qui est la limitation du champ d'application, la PL a été étendu dans le but de supporter d'autres domaines de discours autre que ceux que permettaient d'exprimer les formules logiques. Cette extension venait après une étude théorique, faite par Joxan Jaffar et Jean Louis Lassez en 1986,

qui a fait naissance au concept général de "*Programmation logique par contraintes*" (CLP)[JAF 87]. Le besoin de combiner au processus de déduction logique des algorithmes incrémentaux efficaces de résolution des contraintes appelés "*résolveurs de contraintes*", a placé d'emblée la CLP au carrefour de plusieurs grands domaines, tel que la recherche opérationnelle, l'intelligence artificielle, le calcul formel…etc.

Notre article rentre dans le cadre d'application des principes de la programmation logique par contraintes, et cela afin de résoudre les problèmes informatiques d'une façon simple et efficace. Appliquer la CLP pour résoudre un certain problème, revient à implémenter un résolveur de contraintes ou tout simplement, un algorithme qui donne les solutions du problème décrit par son ensemble de contraintes qui le décrivent. Le problème étudié dans cet article est le puzzle crypte arithmétique : "SEND + MORE = MONEY".

1 Les langages du premier ordre

La programmation logique est basée essentiellement sur les concepts de la logique des prédicats du premier ordre. Cela donne aux programmeurs de ce style, un caractère simple et déclaratif.

L'intérêt principal du calcul des prédicats (ou la logique du premier ordre), par rapport au calcul des propositions, est l'introduction des variables. Il permet ainsi de représenter une large variété d'état du monde réel. Le calcul des prédicats peut être vu comme l'assise mathématique des langages mathématiques.

1.1 Notions de base

Afin de comprendre les aspects de la PL, nous allons dans ce qui suit présenter quelques notions préliminaires de la logique du premier ordre [LIO 84, FAG 96]:

Alphabet :
L'alphabet d'un langage de premier ordre est caractérisée par l'ensemble des symboles suivants :

Les symboles de constantes : a, b, c, …

Les symboles de variables : x, y, z …

Les symboles de prédicats (ou de relations), noté S_p= {P, Q, R …} avec leur arité α, β, γ …

Les symboles de fonction, noté S_f ={f, g, h …} avec leurs arités (Les constantes peuvent être vues, comme des fonctions qui ont l'arité 0).

Les symboles logiques, noté S_l ={$\neg$, $\wedge$, $\exists$} et autres symboles obtenus à partir des abréviations de S_l ($\Rightarrow$, $\vee$, $\perp$, $\Leftarrow$, $\Leftrightarrow$, $\forall$).

Les séparateurs : {',', '(', ')'}.

Termes de premier ordre :
Les termes sont définis inductivement comme suit :

Toute constante est un terme.

Toute variable est un terme.

Si *f* est un symbole de fonction d'arité *n*, et t_1, t_2, … , t_n sont des termes alors f() est un terme.
Remarque : l'ensemble des termes *clos*, noté par $T(S_f)$, est l'ensemble des termes ne contenant pas de variables.

Les formules :
Soit S_p : l'ensemble des prédicats du premier ordre noté p,q, … avec leur arité α. P_a : l'ensemble des propositions atomiques d'ordre 1, $P_a = \{P(M_1,\ldots, M_n) / P\ S_p(p)=n, M_1,\ldots M_n \in T\}$. on définie P, l'ensemble des formules logique notées φ, ϕ, … par :
1- $P_a \subset P$.
2- $\phi \in P \Rightarrow \neg\phi \in P$.
3- $\phi, \varphi \in P \Rightarrow \phi \vee \varphi \in P$.
4- $X \in V, \phi \in P \Rightarrow \exists X\ \phi \in P$.

Les littéraux :
Un littéral est une proposition atomique ou la négation d'une proposition atomique.
Les expressions :
Une expression est soit un terme, un littéral, conjonction ou disjonction de littéraux.
Les clauses : Une clause est une conjonction de littéraux universellement quantifié de la forme :
$\forall X1,\ldots, Xn\ (L1 \vee \ldots \vee Ln)$ noté $\forall(L1 \vee \ldots \vee Ln)ou\{X1, \ldots , Xn\} = \cup V(Li)$, i=1..n.

La sémantique des langages des prédicats du premier ordre est définie par : l'interprétation, l'inconsistance, et la validité des fbf (les fbf permettent d'affecter des vérités a toutes connaissances déjà établies)[FAG 96].

Ce qui intéresse un programmeur, est de résoudre le problème posé d'une manière efficace. Dans ce style de programmation, la résolution est une méthode de démonstration automatique qui utilise un système de déduction pour les formules du premier ordre qui sont sous forme de clauses. L'application de cette méthode nécessite préalablement, soit d'exprimer le problème directement sous forme de clauses, soit de transformer un ensemble de formules de premier ordre en un ensemble de clauses. L'algorithme d'unification permet ensuite d'appliquer la règle de résolution à cet ensemble de clauses [LAS 93].

1.2 L'unification

L'unification est une opération de base dans un programme logique. Pour pouvoir appliquer la règle de résolution à deux clauses, il est nécessaire de savoir si deux ou plusieurs formules atomiques peuvent être unifiées [SUP, LAS 93], i.e. s'il existe une substitution des variables de ces formules par des termes du langage qui permettent de les rendre égales.

L'unification représente le processus par lequel tout langage logique met en correspondance un atome avec un fait, où la tête d'une règle pour vérifier le but

proposé. L'algorithme utilisé pour appliquer ce processus s'appelle algorithme d'unification.

1.2.1 L'unification des termes

Deux termes M et N sont unifiables par σ, si ∃σ tel que Mσ =Nσ (σ est l'unificateur de M et N). On note U (M, N) l'ensemble des unificateurs de M et N. Un unificateur principal de M et N, est un unificateur σ, tel que : ∀ρ∈ U(M, N), ∃θ tel que σθ = ρ.

Exemple :
أ) f(X1, X1) et f(a,b) ne sont pas unifiables.
ب) f(X1, X2) et f(a,X3) sont unifiable avec l'unificateur σ = {X1/a, X2/X3}.

1.2.2 L'algorithme d'unification

Le premier algorithme d'unification sur les termes du premier ordre, fut l'algorithme de Herbrand Robinson [FAG 96], qui est un algorithme non déterministe qui accepte en entrée un système d'équation et retourne en sortie un système d'équation simplifié obtenu par l'application des règles de simplification suivante :

Dec : $f(M1,\ldots, Mn) = f(N1,\ldots, Nn) \wedge \Gamma \Rightarrow M1 = N1 \wedge \ldots \wedge Mn = Nn \wedge \Gamma$.
Dec ⊥: $f(M1,\ldots, Mn) = g(N1,\ldots, Nm) \wedge \Gamma \Rightarrow \perp$ si $f \neq g$ ou $n \neq m$.
Triv : $X = x \wedge \Gamma \Rightarrow \Gamma$.
Var : $X = M \wedge \Gamma \Rightarrow X = M \wedge \Gamma\sigma$ si $X \notin V(M)$, $\sigma = \{X \Leftarrow M\}$.
Var ⊥: $X = M \wedge \Gamma \Rightarrow \perp$ si $X \in V(M)$ et $X \neq M$.

Jusqu'à l'obtention d'une forme irréductible.

Remarque : Quel que soit l'ordre de l'application des règles de simplification, l'algorithme de Herbrand est correct et complet. Une famille d'algorithmes d'unification peuvent donc être dérivées en fixant la stratégie d'application des règles, nous citons à titre d'exemple :l'algorithme de Robinson, l'algorithme de Huet l'algorithme de Martelli-Montanari, l'algorithme de Peterson-Wegman … etc.

2 La programmation logique

La programmation logique est née de la découverte qu'une partie substantielle de la logique du premier ordre pouvait recevoir une interprétation procédurale qui est basée sur la stratégie de résolution SLD (Selected Lineary Defined Resolution [ROB 65]). Cette stratégie fut introduite au début des années 70, est une règle de simplification des buts, elle procède par une unification avec les têtes de clauses du programme.

La programmation logique touche aujourd'hui des champs d'application très variés [SEB 95] tels que :

1- La conception de systèmes experts dans le but de simuler l'expertise humaine.
2- La conception des SGBDR.
3- Le traitement du langage naturel.
4- L'enseignement assisté par ordinateur
5- L'automatisme.
6- La législation.

En outre, d'autres issues théoriques sont basées sur la PL, à titre d'exemple nous citons : Le contrôle de la communication dans les systèmes multi-processus, la transformation et la synthèse de programmes et la programmation parallèle.

Nous présentons dans ces qui suit les principaux avantages de la PL :

1- *Simplicité* : L'aspect déclaratif de la PL offre une manière très simple pour la résolution des problèmes. La tâche du programmeur est réduite à la description des connaissances et du problème à résoudre.
2- *Puissance* : La puissance de la PL réside dans le fait d'utiliser le concept d'unification et de la résolution pour inférer la solution du problème, à partir des descriptions.
3- *Procédures non directionnelles* : Les procédures utilisées dans la PL sont différentes de celles utilisées dans les langages conventionnels au sens ou une procédure peut être utilisée pour résoudre différentes types de problèmes et cela selon l'instanciation de ses arguments.

Cependant la programmation logique connais quelques inconvénients :

1- *Lenteur* : La lenteur de la PL est dû essentiellement à l'inadaptation du style logique vis à vis du modèle de Von-Neumann. Cela répercute irrémédiablement sur le temps d'exécution des programmes logiques.
2- *Pauvreté en représentation des données* : La PL n'est pas adaptée à la définition de nouveau types de structures de données. Les structures existantes sont parfois inefficaces en comparaison avec des structures plus spécialisées telle que les vecteurs par exemple.

Afin de voir l'application pratique des notions vues jusqu'à maintenant, nous allons dans ce qui suit présenter brièvement le langage PROLOG(acronyme de PROgrammation LOGique), qui est le plus représentant des langages de PL.

PROLOG repose sur l'utilisation de la logique du premier ordre dans la résolution des problèmes. Un programme PROLOG comporte une base de *faits* et un ensemble de *règles* :

Base de faits :

Les faits sont des clauses, qui expriment des connaissances qui sont inconditionnellement vraies.

Exemple : les clauses suivantes constituent une base de faits exprimant des connaissances aériennes entre villes, ainsi que les horaires de départ et d'arrivée :
Liaison (Alger, Oran, 1244, 1355).
Liaison (Oran, Ghardaya, 1512, 1723).
Liaison (Ghardaya, Tamanraset, 1833, 1900).
Tel que 1244 par exemple signifie $12^{h}44^{mn}$.

Règles:

Les règles permettant de définir, sous forme de clauses complètes, de nouveaux concepts à partir de faits ou de concepts préexistants. L'emploi des règles est particulièrement économique en place mémoire.
Exemple : Les règles suivantes sont des règles exprimant des relations entre les individus de la base de faits :

Lien (Ville_depart, Ville_arrivé, Heur_depart, Heur_arrivé) ←
Liaison (Ville_depart, Ville_inter, Heur_depart, Heur_inter),
Liaison (Ville_inter, Ville_arrivé, Heur_inter, Heur_arrivé) .

Questions :

La base des faits et l'ensemble des règles constituent un programme dont on peut demander l'exécution en posant des questions. Exemple : La question :
"? Liaison (Alger, Oran, 1244, 1355)" est une question, qui aura la réponse "Vrai". La réponse à la question "? Liaison (Oran, X, 1512, 1723)", consiste à trouver la valeur de X, qui rend le fait Liaison (Oran, X, 1512, 1723) vrai. L'inconnue X, sera liée (ou instanciée) à une constante (Ghardaya).

La résolution en PROLOG, revient à explorer la base de faits afin de répondre à la question posée. Elle peut être faite suivant différentes stratégies.

3 La programmation logique avec contraintes

Nous allons discuter dans ce qui suit l'introduction des contraintes dans la programmation logique pour la résolutions des problèmes. Prenons par exemple le langage : PROLOG, Le langage PROLOG a été conçu initialement, dans le but de traiter les langages naturels [COH 90]. L'application de PROLOG a montré ses limites de ce qui concerne le traitement des opérations arithmétique. Par exemple le but 'S+M=O' génère en PROLOG un échec, car les opérateurs '+' et '=' ne sont pas considérés comme des symboles valides de fonction et par conséquent, l'unification ne peut pas être faite.

L'objectif de la programmation logique avec contraintes, est la résolution d'un ensemble de contraintes. Les contraintes sont spécifiques au problème posé, elles doivent le décrire pour qu'on puisse appliquer le principe de la CLP afin de le

résoudre. On peut dire qu'un certain isomorphisme doit exister entre le problème posé et les contraintes qui le décrient.

La CLP représente une manière puissante et efficace pour résoudre plusieurs problèmes, par exemple : l'assignation des tâches, le classement de voitures, la coloration des graphes.

Un programme logique avec contraintes, est un ensemble fini de clauses de la forme : A ←C1, C2,…, Cm | A1, A2, … , An.
Ou Les C_i, sont des contraintes et les A_i sont des atomes. Ces clauses ont un sens déclaratif. A est vrai, si C1, C2,…, Cm, A1, A2, … , An sont vraies. Une interprétation procédurale est la suivante : Pour montrer A, il suffit de satisfaire C1, C2,…, Cm, et montrer A1, A2, … , An. Un but CLP, est une clause sans littéral positif, de la forme :
$\neg C_1 \vee \ldots \vee \neg C_m \mid \neg A_1 \vee \ldots \vee \neg A_n$
que l'on note : ? - $C_1,\ldots,C_m \mid A_1,\ldots,A_n$.

3.1 Les contraintes

Une contrainte est une formule logique atomique (relation), qui est interprétée dans une structure particulière et non pas dans une interprétation de Herbrand comme le sont, par exemple les prédicats en programmation logique classique.

Les variables appartenant à une contrainte, ne doivent pas être des caractères ou des listes [COD 95]. Par exemple si on considère la contrainte arithmétique {D+E=Y}, D, E et Y doivent appartenir à un domaine numérique tel que les entiers ou le réelles…etc.

Une contrainte C est satisfaite, si elle admet une solution dans la structure S qu'on note $S \models C$ (S est une structure mathématique tel que 3, ∠,…etc.). Une conjointe de contraintes forme ce qu'on appelle par *système de contraintes*.

3.2 Domaine de contraintes

Les programmes appelés CLP(X), sont des programmes paramètrés par la donné de la structure X, qui fixe le domaine de satisfaction des contraintes, tels que les domaines des entiers, booléens, réelles, rationnels … etc.

3.2.1 Les domaines finis

Ce domaine est introduit pour améliorer l'efficacité des programmes CLP(X) par la recherche en intelligence artificielle, où un domaine fini est tout simplement un ensemble de valeurs numériques ou symboliques de cardinalité fini, par exemple : {1, 5, 7}, [0..9] ou {astronomie, physique, mathématique}.

Les contraintes existantes dans ce domaine, sont des contraintes linéaires et des contraintes symboliques qui ne relèvent pas des opérations mathématiques habituelles. Par exemple la relation *Existe* (valeur, [X1 | X]) qui retourne la valeur vraie, si valeur

existe dans la liste [X1 | X]. Les domaines finis sont utilisés pour résoudre plusieurs types arithmétiques tels que (=, ≥, ≤, <, >, ≠) entre des termes d'application industrielle comme les problèmes combinatoires [VAN 89], la simulation des circuits, l'aide à la décision…etc.

Dans ce domaine, la résolution des contraintes est basée sur des méthodes appelées technique de consistance. Ces techniques dérivées d'une autre branche de l'intelligence artificielle : problèmes de satisfaction des contraintes (PSC). Un PSC consiste en un ensemble de variables, un domaine fini et discret pour chaque variable et un système de contraintes entre ces variables. L'approche adopté par ces techniques est différente de celle adoptée par les algorithmes de résolution de contraintes dans le sens où elle sont incapables de prouver la satisfiabilité d'un système de contraintes, avant que toutes les variables soient instanciées à des valeurs exactes. Le problème que nous allons détailler par la suite consiste le plus populaire des exemples qui illustrent les domaines finis.

4 Le problème " Send + More = Money"

L'exemple du puzzle crypte arithmétique : "SEND+MORE=MONEY" est souvent pris comme exemple d'application [FRA 90, PAS 91], pour la résolution de problèmes en se basant sur la CLP. Ce problème consiste à trouver une valeur au lettres : {S, E, N, D, M, O, R, Y} parmi les chiffres {0,1,…, 9} de telle manière que les chiffres attribués soient différents, et tel qu'on a :

```
   S  E  N  D
 + M  O  R  E
-------------
 M  O  N  E  Y
```

Cette somme est équivalente à l'égalité suivante :

(D + 10N + 100E + 1000S) + (E + 10R + 100O + 1000M) =
Y + 10E + 100N + 1000O + 10000M

Ce puzzle peut être exprimé par :

```
sendc ([S, E, N, D, M, O, R, Y]) :- domain ([S, E, N, D, M, O, R, Y],[0..9]),
                                    alldiferent ([S, E, N, D, M, O, R, Y]),
                                     noteq (S, 0),
                                     noteq (M,0),
eqln (( D + 10N + 100E + 1000S ) + ( E + 10R + 100O + 1000M ) =
Y + 10E + 100N + 1000O + 10000M).

alldiferent([]),
alldiferent([X | Xs]) :- outof(X, Xs),
                         alldiferent(Xs).
```

```
outof(X, []).
outof(X, [Y | Ys]) :- X≠Y,
                      outof(X, Ys).

noteq(X,Y) :- X≠Y.
eqln(E1,E2):- E1=E2.
domain([],[_,_]).
domain([x | X],[BI,BS]):- x ≥ B1,
                          x ≤ BS,
                          domain(X,[BI,BS]).
```

La contrainte symbolique alldiferent([S, E, N, D, M, O, R, Y]), proposée dans le langages CHIP (Constrainte Handling In Prolog) [FRA 90, PAS 91], signifie que les valeurs de S..Y, sont différents.

Pour résoudre le puzzle, il suffit de poser la question "? sende(L)" au système.
L'algorithme de résolution d'un problème donné, revient à trouver les valeurs des lettres {S, E, …, Y} qui vérifient le but voulu en respectant les contraintes données.

Tout algorithme de résolution basé sur des règles à appliquer et un but à atteindre peut être décrit comme suit [SUP] :

Algorithme Résolution**;**

Début
Commencer avec le but courant
Tant que (le but courant n'est pas vide) **faire**
    Choisir le sous but le plus à gauche;
    Si (une règle est applicable)
    Alors
        Sélectionner la première règle applicable;
        Former un nouveau but; (***qui de vient le but courant**)
    Sinon
        Retour arrière; (***Back track**)
    Fsi;
Fait;
Fin.

En introduisant la notion d'unification, on aura

Algorithme Résolution**;**
Début
Commencer avec le but courant
Tant que (le but courant n'est pas vide) **faire**
    Soit le but courant $G_1,\ldots, G_k$; (*k≥1*)
    Choisir le sous but le plus à gauche G_1
    Si (une règle est applicable à G_1)
    Alors
        Sélectionner la première règle $A \leftarrow B_1,\ldots, B_j$; (*j≥0*)
        Soit δ l'unificateur de G_1 et A;
        Le but courant de vient : $B_1\delta,\ldots, B_j\delta, G_2\delta, G_3\delta,\ldots, G_k\delta$;
    Sinon
        Retour arrière;
    Fsi;
Fait;
Fin.

Les résultats du puzzle "SEND+MORE=MONEY" ainsi que les approches utilisées et leurs évaluation, sont détaillés dans la section 5.3.

5 Mise en œuvre et résultats

5.1 Introduction

La programmation logique avec contraintes est l'extension de la programmation logique habituelle, et cela en introduisant la notion de contraintes. Cette dernière résout plusieurs problèmes qui n'ont pas de solution en 'Prolog' par exemple. Les contraintes décrivent implicitement les relations qui existent entre les objets, et couvrent des riches domaines de calculs tels que : les entiers, les réels, les rationnels,… etc.

Plusieurs langages CLP existent, la différence entre ces langages, réside principalement dans le domaine de calcul :

Le langage CHIP (Constrainte Handling In Prolog) [FRA 90, PAS 91], développé par le centre ECRC (European Computer industry Research Center), comporte trois domaine de calcul: les booléens, les rationnels et les domaines finis, qui servent à modifier l'espace de recherche utilisé en programmation logique.

CAL (Contraintes Avec Logique) [FRA 90] est un autre langage CLP dont le principal apport est de pouvoir manipuler des contraintes non linéaires en se basant sur l'algorithme de Grobner [FRA 90].

Développé par l'université d'Aix-Marseille (France) [COL 90], Prolog III est un langage CLP inclut trois domaines de calculs: les rationnels arithmétiques, les booléens et les listes finies. La syntaxe de ce langage diffère légèrement de celle des langages CLP. Une règle a la forme : $T_0 \rightarrow t_1,\ldots, t_n,C$, où : les t_i sont des termes, et C est une liste de contraintes. Un état de calcul est de la forme : $(W, t_1,\ldots, t_n,C)$ où: W: un ensemble de variables. t_i : des termes à utiliser et C : la liste de contraintes à satisfaire.

Les langages CLP, permettent de résoudre des problèmes qu'on les décrit sous forme de contraintes à respecter, et un but à atteindre. La mise en œuvre de tels langages est basé sur l'implémentation d'un résolveur de contraintes, qui n'est autre qu'un ensemble d'algorithmes retournant comme résultats les solutions du problèmes. Ces solutions, doivent vérifier le but voulu et appartenir au domaine du calcul.

Dans ce qui suit nous allons décrire l'application que nous avons réalisé pour résoudre le problème : "SEND+MORE=MONEY". Le but de cette réalisation est de voir l'apport de la programmation logique avec contraintes dans la résolution de problèmes par rapport à la programmation habituelle ; et de montrer également comment avec cette nouvelle vision, on peut formaliser et résoudre des problèmes d'une manière simple et efficace.

5.2 L'application "CLPA"

Notre application, qu'on appel CLPA (Constrainte Logic Programing Application), est réalisée dans le but de résoudre le puzzle : "SEND+MORE=MONEY". Pour cela nous avons implémenté un résolveur de contraintes, basé sur le principe de la programmation logique avec contraintes. Dans l'implémentation, nous nous sommes basé sur notre algorithme introduit dans la section 4, qui permet de trouver toutes les solutions possibles du problème précédent. L'application CLPA permet de résoudre le puzzle, en changeant les contraintes et avec des strategies de résolution différente inspiré du domaine de l'intelligence artificielle. Notre application permet aussi de résoudre le problème du "Carré magique", et cela dans le but de montrer que le principe de la CLP peut être appliquer à n'importe quel problème, qui peut être décrit par un ensemble de contraintes à respecter, et un objectif à atteindre.

L'application CLPA, a été réalisée sous une plate forme WINDOWS sous Borland Delphi professionnel, version 4.0 (build 5.37). Le choix de cet environnement est dicté par la souplesse , assurée par Delphi, de traduire les algorithmes sous forme de programmes compréhensible par la machine; sans oublier bien sûr la convivialité et le principe de la programmation basé sur les événements, offerts par cet environnement.

5.2.1 Les approches utilisées

Pour assurer l'efficacité de la résolution, il est nécessaire de définir des stratégies et des choix.

L'approche utilisée par notre application, afin de résoudre le puzzle crypte arithmétique "SEND + MORE = MONEY", est basée sur stratégie d'exploration en profondeur d'abord (*depth first*). Nous n'avons pas choisit la stratégie en largeur d'abord (*breat first*) pour éviter son principal inconvénient, qui réside dans l'encombrement de mémoire et du temps d'exécution.

Notre stratégie consiste à choisir un chemin ,en respectant les contraintes, jusqu'à l'obtention d'un succès ou d'un échec; et de prendre ensuite un autre choix.

Dans notre implémentation du résolverur de contraintes, nous nous sommes basés sur le concept de récursivité, ce concept permet de faciliter la tâche de résolution et de la simplifier. Le problème de saturation de pile ne se pose pas dans notre cas, puisque le nombre d'appels, dans notre cas, est réduit (égal au nombre de variables, i.e. huit).

Notre application CLPA utilise, dans le cas où la contrainte alldiferent ([S, E, N, D, M, O, R, Y]) devrait être vérifiée, trois algorithmes :

Algorithme 1 : Dans cet algorithme, l'arbre de dérivation est exploité avec la méthode connue en intelligence artificielle par "la recherche aveugle". La vérification du but, se fait au niveau des feuilles.
Algorithme 2 : Dans cet algorithme, l'exploration de l'arbre se fait en vérifiant à chaque fois si la lettre qu'on doit choisir, n'a pas la valeur déjà affectée à une lettre déjà visitée. En d'autres termes, la contrainte "alldiferent ([S, E, N, D, M, O, R, Y])", est subdivisée en sous contraintes que les on vérifie à chaque avancement dans l'arbre de dérivation.
Algorithme 3 : Dans cet algorithme on applique la même chose que dans l'*algorithme2*. En plus de cela, et de la même manière que dans l'algorithme précédent, le but " eqln ((D + 10N + 100E + 1000S) + (E + 10R + 100O + 1000M) =
Y + 10E + 100N + 1000O + 10000M)" est subdivisé en sous buts, qui sont testé à chaque fois qu'on avance dans l'arbre. La subdivision du but, est fait d'une manière *astucieuse*, qui permet de rajouter un sous but à chaque choix d'une nouvelle lettre.

Afin de comparer les résultats en variant les contraintes du puzzle, la CLPA permet aussi de résoudre le puzzle en omettant la contrainte "alldiferent ([S, E, N, D, M, O, R, Y])". En plus de cela, et dans le but de montrer que nos approches peuvent être utilisées pour résoudre d'autres problèmes, qu'on décrit leur spécificité à l'aide des contraintes et du but voulu, l'application CLPA permet aussi de résoudre le problème de carré magique d'ordre trois. Le résolveur de contrainte de ce problème à la même structure que celui utilisé pour résoudre le puzzle arithmétique.

5.3 Résultats et discussion

L'étude faite dans cet article nous permet de voir clairement l'intérêt de la programmation logique par contraintes, pour la résolution des problèmes. La CLP

permet de résoudre un problème en spécifiant seulement l'ensemble des contraintes qu'il doit respecter et le but qu'on doit atteindre.

Notre application "CLPA", qui est une application pratique des concepts de la CLP, permet de résoudre le puzzle "SEND+MORE=MONEY". En se basant sur le fait que le contrôle des programmes logiques, est dicté par le choix d'ordre de but et de règle, notre application propose à l'utilisateur trois méthodes différentes, et cela selon l'algorithme appliqué.

De point de vue temps d'exécution, l'*algorithme 1*, prend plus de temps pour trouver la solution. Cela est logique, car l'exploration de l'arbre de dérivation est aveugle, i.e. on n'utilise aucune intelligence dans la recherche des solutions. Toutes les branches de l'arbre sont visitées et le résolveur prend la trace qui vérifie l'objectif voulu.

L'*algorithme 2*, améliore le temps d'exécution. Cette amélioration est due au test qui se fait à chaque choix de nouvelle valeur dans l'arbre. L'*algorithme 3* est le meilleur algorithme de point de vue temps de réponse. L'amélioration considérable du temps d'exécution est une conséquence de la décomposition astucieuse du but et de la contrainte "alldiferente". La décomposition du but permet d'éliminer plusieurs branches de l'arbre de dérivation ce qui accélère l'exécution. L'élimination est plus intéressante quand elle est faite à un petit ordre de l'arbre, cela nous permet de conclure le résultat suivant :

Si ℜ (but_final), est un arrangement composé de sous-but du but final. ℜ donne un meilleur temps de réponse, avec notre résolveur de contraintes, s'il permet de balayer le but final en une profondeur minimale de l'arbre d'exploration.

L'application CLPA, permet aussi de résoudre le problème de carré magique, ce dernier représente aussi un bon exemple pour la résolution basée sur la CLP. Notre résolveur de contraintes permet de donner les solutions du carré magique d'ordre trois. Notons que dès que la taille du carré magique dépasse trois, le temps de réponse devient énorme. Cela est dû au fait que l'arbre de dérivation devient très volumineux pour trouver toutes les solutions, c'est pour cela que l'application propose l'option de résolution qui donne qu'une seule solution, cette dernière est la première solution rencontrée.

Dans le cas ou les contraintes du puzzle arithmétique incluent la contrainte: "alldiferent ([S, E, N, D, M, O, R, Y])", l'ensemble de solutions trouvées par notre résolveur est :

{1 5 6 3 2 8 7 0, 1 7 8 3 4 6 5 0, 1 8 9 2 6 4 3 0, 1 8 9 5
3 7 6 0, 2 4 6 3 1 9 8 0, 2 5 7 4 1 9 8 0, 2 7 9 1 6 4 3 0,
2 7 9 3 4 6 5 0, 3 6 9 4 2 8 7 0, 3 8 1 5 2 7 6 0, 4 3 7 2
1 9 8 0, 4 5 9 2 3 7 6 0, 4 5 9 3 2 8 7 0, 5 4 9 1 3 7 6 0,
6 3 9 1 2 8 7 0, 7 5 2 6 8 0 9 1, 7 8 5 1 6 3 2 0, 9 4 3 1

2 7 6 0, 9 4 3 2 1 8 7 0, 9 5 4 3 1 8 7 0, 9 6 5 4 1 8 7 0, 9 7 6 1 5 4 3 0, 9 8 7 1 6 3 2 0, 9 8 7 2 5 4 3 0, 9 8 7 4 3 6 5 0}.

Où chaque élément de l'ensemble est une solution du puzzle, qui affecte une valeur au vecteur [D, E,Y, N, R, O, S, M]. Le nombre de solution est alors égal à 25. Si on rajoute seulement la contrainte M≠0, on aura une seule solution qui est : [D, E,Y, N, R, O, S, M] ≡ [7, 5, 2, 6, 8, 0, 9, 1]. On peut déduire que tant qu'on augmente le système avec plus de contraintes, on aura un nombre de solutions qui soit inférieur ou égal au nombre précédent. En effet, dans le cas ou la contrainte "alldiferent" n'est pas introduite, le nombre total de solutions possible est égal à 1155.

6 Conclusion

La programmation logique avec contraintes, a atteint aujourd'hui une maturité industrielle qui se traduit par l'existence de produits logiciels commerciaux, citons par exemple : Prolog III, Prolog III (prologia), CHIP (cosytec), et par celle d'un nombre substantiel d'applications opérationnelles.

Les langages logiques constituent une autre alternative aux langages impératifs. Ces langages possèdent une base théorique mathématique qui est la théorie des prédicats.

L'intérêt du style logique réside dans la manière de résolution des problèmes; en utilisant ce style, le programmeur décrit les connaissances nécessaires à la résolution du problème posé sous forme de clauses. L'interpréteur cherche par la suite, l'ensemble des solutions du problème en utilisant les mécanismes de résolution et d'unification.

Dans notre article, nous avons étudié le style de programmation logique avec contraintes avec ses différents concepts de base. Afin de sentir mieux l'apport de la CLP, nous avons appliqué les approches étudiées pour la résolution du puzzle crypte arithmétique "SEND+MORE=MONEY", et aussi au problème des carrées magiques. Trois algorithmes sont proposés et évalués, les résultats obtenus par notre application "CLPA" sont ensuite discutés. Notre application CLPA peut être améliorée, dans le but de traiter une classe de problèmes plus vaste. Cela est faisable, car le principe de résolution est le même, il suffit de changer le domaine de calcul, et d'introduire les nouvelles contraintes.

Comme nous avons noté, la résolution basée sur CLP, peut induire un temps d'exécution considérable, mais n'oublions pas qu'avec ce style de programmation le travaille du programmeur est presque court-circuité, ce qui représente un gain énorme de temps et d'effort. Une perspective de cet article sera d'utiliser le concept des heuristiques dans la recherche des solutions.

Remerciement

Nous tenons à remercier Mme C. IGHILAZA pour l'aide et les conseils qu'il nous a prodigués durant l'élaboration de cet article.

Références bibliographiques

[COD 95] Philipe Codognet
Programmation logique avec contraintes : Une introduction.
Technique et science informatique. Vol. 14, N. 6, pp 662-692, 1995.

[COH 90] Jaques Cohen.
Constraint logic programming language.
Communication of the ACM, Vol 33 ; no 7 ; p 52-68. 1990.

[COL 90] A Colmerauer
An introduction to Prolog III
Communication of the ACM, Vol 33, N° 7, pp 70-90, July 1990.

[CON 86] M. Condillac.
Prolog, fondements et application.
Edition Dunod, Paris 1986.

[FAG 96] François Fages
Programmation logique par contraintes.
Ecole polytechnique, Garnes, Janvier 1996.

[FRA 90] Frank Kriwaczed
An Introduction to logic programming
Departement of computing , Imperial College London, England 1990.

[JAF 87] J. Jaffar, J. L. Lassez
Constraint logic programming.
Research report, University of Melbourne1986, Alsoin the proceeding of POPL'87, 1987.

[KOW 74] R. Kowalski
Predicat logic as programming language.
Information processing 74, pp 569-574, 1974.

[LAS 93] Lassaigne R, De Rougemont M
Logique et fondement de l'informatique.
Edition Hermes, 1993.

[LIO 84] D. W. Lioyd
Fondation of logic programming
Pring verlof, Edition Eyrolles, 1984.

[PAS 91] Pascal Van Henteryck
The CLP language CHIP: Constraint solving and applications
Brown University, Ch2961, IEEE , January 1991.

[ROB 65] Robinson A.
A machine oriented logic based on the resolution principle.
O'Reilly and Association, Inc. 103 Morris Street, Suite A Sebastopol, CA 95472, 1965.
[SEB 95] Sebesta, Robert W.
Concept of programming languages.
Edition Addisson Wesley, 1995.
[SUP] Support du cours du module "Programmation avancée", assuré par Dr. Belkhir Abdelkader,
Institut d'informatique. USTHB. 1999/2000.
[VAN 89] Van P. Hentenryck
Constraint Satisfaction in logic programming.
MIT Press, 1989.

Annexe : Algorithmes utilisés

Cet annexe donne les principaux algorithmes utilisés par l'application CLPA. Les algorithmes sont écrits en langage PASCAL.

```
function objectif1 :boolean;
begin
if(1000*CAR[7]+100*CAR[2]+10*CAR[4]+CAR[1] +
1000*CAR[8]+100*CAR[6]+10*CAR[5]+CAR[2] =
10000*CAR[8]+1000*CAR[6]+100*CAR[4]+10*CAR[2]+CAR[3])then begin
objectif1:=true;  end
else objectif1:=false;
end;

function objectif2 :boolean;
begin
if((1000*CAR[7]+100*CAR[2]+10*CAR[4]+CAR[1] +
1000*CAR[8]+100*CAR[6]+10*CAR[5]+CAR[2] =
10000*CAR[8]+1000*CAR[6]+100*CAR[4]+10*CAR[2]+CAR[3])and (dif)) then
begin objectif2:=true;  end
else objectif2:=false;
end;

// les contraintes qui lient les variables
procedure contraintes;
begin
if(alldiferent)then begin if(objectif2)then affichage end
 else if(objectif1)then affichage;
end;
```

```
procedure trying1 (level:integer );
var i:integer;
begin
for i:=0 to 9 do // 0..9 le domaine des variables
 begin
   if(i_diff(i,level))then //i_diff(i,level) : si i est déjà utilisé ce n'est pas la paine.
   begin
    CAR[level]:=i;
    if(level=L)then contraintes
    else trying1 (level+1);
   end;
 end;
if(level=1)then form1.Memo1.Lines.Add('Fin de recherche de solutions(
'+inttostr(nbr_solution)+' solutions trouvées).');
end;

procedure trying2 (level:integer );
var i:integer;
begin
for i:=0 to 9 do // 0..9 le domaine des variables
 begin

    CAR[level]:=i;
    if(level=L)then contraintes
    else trying2 (level+1);

 end;
if(level=1)then form1.Memo1.Lines.Add('Fin de recherche de solutions(
'+inttostr(nbr_solution)+' solutions trouvées).');
end;

function contrainte(level:integer): boolean;
var s:integer;
begin
case level of
1: contrainte:=true;
2: contrainte:=true;
3: begin
   s:=CAR[1]+CAR[2];
   if(( (s-10*trunc(s/10)))=CAR[3])then contrainte:=true else
contrainte:=false;//D+E=Y [avec une eventuelle retenue ]
   end;
4: contrainte:=true;
5: begin
   s:=(10*CAR[4]+CAR[1])+(10*CAR[5]+CAR[2]);
   if((10*CAR[2]+CAR[3])=s-100*trunc(s/100))
```

```
   then contrainte:=true
   else contrainte:=false;
   end;
6: begin
   s:=(100*CAR[2]+10*CAR[4]+CAR[1]+100*CAR[6]+10*CAR[5]+CAR[2]);
   if (100*CAR[4]+10*CAR[2]+CAR[3])=s-1000*(trunc(s/1000)) then
  contrainte:=true
  else contrainte:=false;
   end;
7: contrainte:=true;
8: if(1000*CAR[7]+100*CAR[2]+10*CAR[4]+CAR[1] +
1000*CAR[8]+100*CAR[6]+10*CAR[5]+CAR[2] =
10000*CAR[8]+1000*CAR[6]+100*CAR[4]+10*CAR[2]+CAR[3])then
contrainte:=true
else contrainte:=false   ;

end;
end;

procedure trying3(level:integer );
var i:integer;
begin
for i:=0 to 9 do // 0..9 le domaine des variables
 begin

   CAR[level]:=i;

   if(i_diff(i,level))and (contrainte(level))then
   begin
     if(level=L)then affichage
     else trying3 (level+1);

   end;
 end;
 if(level=1)then form1.Memo1.Lines.Add('Fin de recherche de solutions(
'+inttostr(nbr_solution)+' solutions trouvées).');
end;

procedure trying_1(level:integer);
var i:integer;
begin
for i:=0 to 9 do // 0..9 le domaine des variables
 begin

    CAR[level]:=i;
    if(level=L)then contraintes
```

```
    else trying_1 (level+1);

 end;
if(level=1)then form1.Memo1.Lines.Add('Fin de recherche de solutions(
'+inttostr(nbr_solution)+' solutions trouvées).');
end;
```