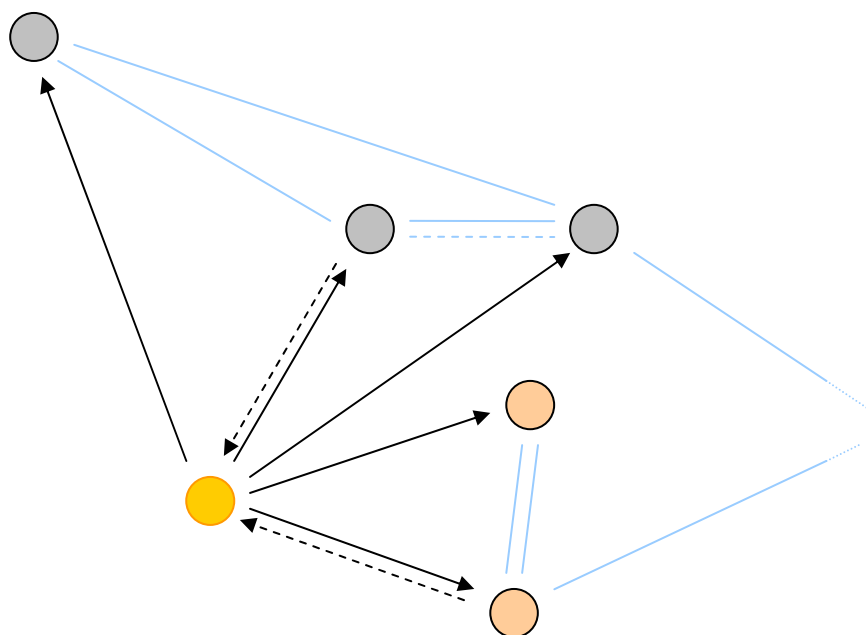


RAPPORT DE PROJET

Travail d'Etude et de Recherche de Maîtrise Informatique



REALISATION D'UNE PLATEFORME DISTRIBUEE POUR LES RESEAUX DE CONTRAINTES

par

Joël GIRARDEAU – Paul ROUAULT – Rémi SOULAGE – Benjamin TABARIÈS

JANVIER - AVRIL 2004

Encadrants : Christian BESSIÈRE – Rémi COLETTA – Arnold MAESTRE

Sommaire

1. Introduction	4
2. Etat de l'art : les réseaux de contraintes	5
2.1. Résolution d'un problème centralisé	5
2.2. Résolution d'un problème distribué	9
3. Recherche	11
3.1. Cahier des charges	11
3.1.1. Objectifs	11
3.1.2. Niveaux d'utilisations	11
3.1.2.1. Simple utilisateur	11
3.1.2.2. Chercheur	11
3.2. Outils	12
3.2.1. Matériel	12
3.2.2. Langage de programmation	12
3.2.3. Interface de communication entre chaque processus	12
3.2.3.1. Madkit	12
3.2.3.2. PVM	13
3.2.3.3. MPI	13
3.2.3.4. Socket	14
3.2.4. Générateur d'instances	14
3.2.4.1. Utilisation du générateur	14
3.2.4.2. Exemple d'un fichier généré	17
3.2.5. Solveur	18
3.2.5.1. Possibilités	18
3.2.5.2. Essais	19
3.2.5.3. Solveur choisit	19
4. Développement	21
4.1. Architecture générale de la plateforme	21
4.2. Construction du problème	23
4.3. Incorporation du solveur dans la plateforme	24
5. Résultats	25
6. Discussion	26
7. Conclusion	27

1. Introduction

Afin d'appliquer les méthodologies et les notions enseignées jusqu'en Maîtrise Informatique à l'Université Montpellier II, nous devons réaliser un Travail d'Etude et de Recherche durant 3 mois. Celui-ci nous permet à nous, étudiants, de nous initier à la recherche, d'appliquer les connaissances acquises durant notre scolarité et de favoriser le travail en groupe encadré par de futurs enseignants-chercheurs.

Le projet que nous devons réaliser est une plateforme distribuée pour les réseaux de contraintes. Elle doit permettre, lorsqu'elle sera terminée, de tester différents algorithmes de résolution de problèmes de satisfaction de contraintes, appelés plus généralement CSP (Constraint Satisfaction Problems), en les distribuant sur un ensemble de machines. Cette plateforme pourra être utilisée soit par un chercheur, qui insèrera et testera ses algorithmes, soit par un utilisateur lambda pour résoudre un problème.

Afin de comprendre la démarche que nous avons utilisée pour mener ce projet à son terme, notre rapport se structure de la façon suivante :

Tout d'abord, dans une première partie nous présentons le cadre général de notre projet, c'est-à-dire ce qui existe et ce que notre projet va apporter. Puis dans une seconde partie, nous présentons le travail d'étude et de recherche que nous avons effectué, en commençant par définir le cahier des charges. Ensuite dans une troisième partie, nous expliquons comment nous avons développé la plateforme avant que dans une quatrième partie nous décrivions les résultats obtenus. Enfin, dans la dernière partie, nous discutons de la concordance de nos résultats par rapport à nos objectifs initiaux.

2. Etat de l'art : les réseaux de contraintes

Un réseau de contraintes est un ensemble de contraintes portant sur des variables ayant un domaine fini et discret. Chaque contrainte limite les combinaisons de valeurs que peuvent prendre les variables sur lesquelles elle pèse.

Il existe deux façons de résoudre un problème, soit en le résolvant sur une seule machine, soit sur un ensemble de machines.

2.1. Résolution d'un problème centralisé

La programmation par contraintes est basée sur le modèle déclaratif, c'est-à-dire que l'on énoncera le problème sans spécifier la méthode pour le résoudre. Elle va permettre la résolution de problèmes combinatoires tel le problème ci-dessous :

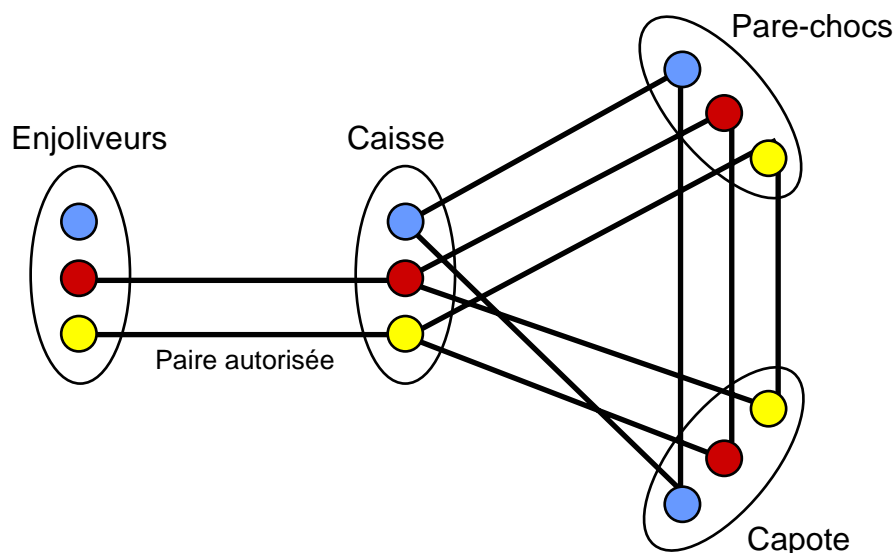


Fig. 1 – Exemple d'un CSP

Une usine fabrique des voitures constituées d'enjoliveurs, d'une caisse, de pare-chocs et d'une capote (ce sont les variables). Toutes ces pièces peuvent avoir la couleur bleu, rouge ou jaune (c'est le domaine des variables). Cependant, il existe des contraintes. Par exemple, lorsqu'il y a une caisse de voiture de couleur rouge, les enjoliveurs devront obligatoirement être rouge.

Une fois le problème défini, on souhaite le résoudre. En outre, le problème de satisfaction de contraintes consiste à affecter à chacune des variables X du réseau une valeur de son domaine $D(X)$ de façon à ce qu'aucune combinaison de valeur non autorisée par les contraintes C n'apparaisse. Ceci constitue un problème NP-difficile.

Une **affectation** peut être soit **totale**, c'est à dire qu'elle instancie toutes les variables du problème, soit **partielle** pour le cas contraire.

Par exemple, $A1 = \{(enjolveurs, rouge), (caisse, bleu), (capote, jaune), (pare-chocs, rouge)\}$ est une affectation totale tandis que $A2 = \{(caisse, bleu), (capote, jaune)\}$ est une affectation partielle.

Une affectation A **viole** une contrainte C_k si toutes les variables de C_k sont instanciées dans A , et si la relation définie par C_k n'est pas vérifiée pour les valeurs des variables de C_k définies dans A .

Sur l'exemple, l'affectation partielle $A2 = \{(caisse, bleu), (capote, jaune)\}$ viole la contrainte 'si la capote est bleue alors la caisse est bleue ou inversement'.

On parlera d'**affectation consistante** si elle ne viole aucune contrainte, et d'**affectation inconsistante** si elle viole une ou plusieurs contraintes.

Sur notre exemple, l'affectation partielle $\{(caisse, bleu), (capote, bleu)\}$ est consistante, tandis que l'affectation partielle $\{(caisse, bleu), (capote, jaune)\}$ est inconsistante.

Une solution est une **affectation totale consistante**, c'est-à-dire une valuation de toutes les variables du problème qui ne viole aucune contrainte.

Sur notre exemple, il n'existe pas d'affectation totale consistante, donc il n'y a pas de solutions.

Il existe plusieurs méthodes de résolution, comme par exemple la résolution naïve appelée aussi 'Generate and Test' :

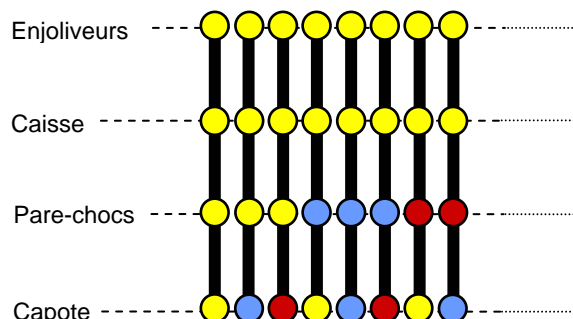


Fig. 2 – Résolution d'un CSP par la méthode 'Generate and Test'

Ainsi, la méthode consiste à générer tous les cas puis à ne garder que les instanciations complètes qui vérifient les contraintes. L'inconvénient de cette méthode est que la complexité du problème (et par conséquent le temps de résolution du problème) est très importante.

De ce fait, il existe des méthodes plus optimisées, tel la méthode du backtracking (« retour en arrière ») qui permet de résoudre un CSP en sortant petit à petit une solution partielle (qui tend vers une solution complète) qui indique des valeurs consistantes pour certaines des variables, en choisissant à plusieurs reprises une valeur pour une autre variable consistante aux valeurs dans la solution partielle courante :

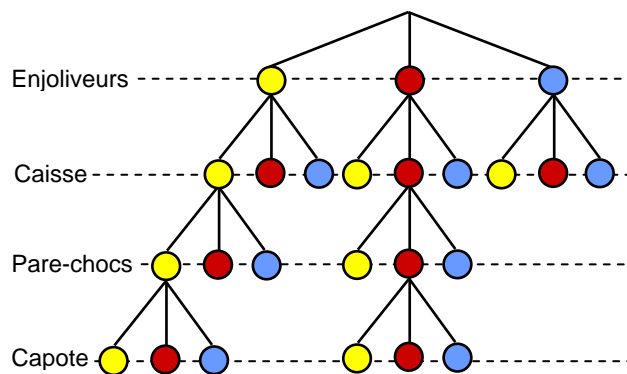


Fig. 3 - Résolution d'un CSP par la méthode du 'backtracking'

La complexité a diminué par rapport à la méthode précédente mais on s'aperçoit qu'une optimisation aurait pu être faite. Ainsi, pour la résolution du problème, il est possible, avant d'exécuter un algorithme de résolution, de réduire l'espace de recherche.

En quelque sorte dans notre exemple, nous aurions pu supprimer la couleur bleue de l'enjoliveur puisqu'elle n'est compatible avec aucune couleur de caisse. De plus, on s'aperçoit que l'affectation de la couleur bleue à une caisse est inconsistante puisque la variable 'caisse' n'est reliée à aucune couleur d'enjoliveur. Puis, la suppression de cette couleur va engendrer l'inconsistance de la couleur bleue pour les pare-chocs et la capote. Ainsi, on obtiendra un problème réduit :

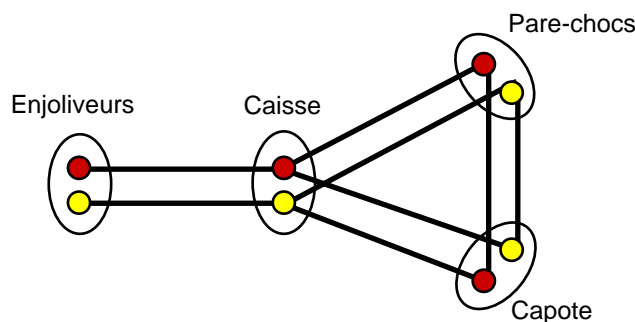


Fig. 4 - Exemple d'un CSP réduit

Ainsi, après avoir réduit l'espace de recherche, en appliquant la méthode du backtracking, on diminue sensiblement la complexité de notre problème :

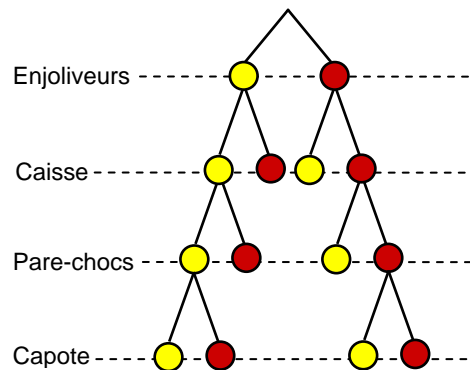


Fig. 5 – Résolution d'un CSP réduit par la méthode du 'backtracking'

Dans la suite, toute cette phase de résolution d'un CSP centralisé sera masquée par l'outil nommé *solveur*.

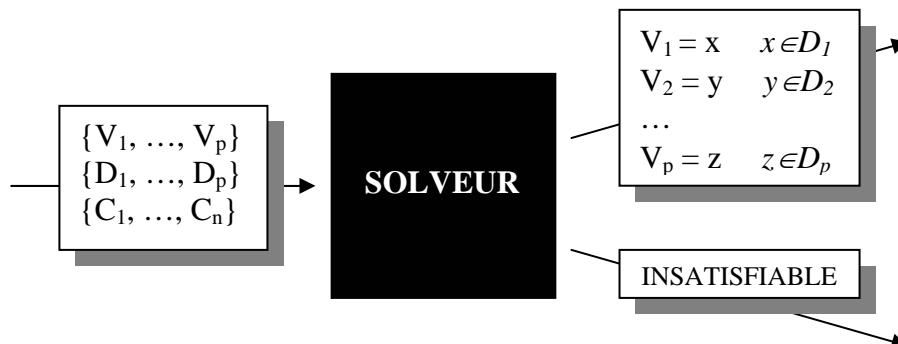


Fig. 6 - Fonctionnement d'un solveur

Le fonctionnement général d'un solveur est par conséquent simple. Comme présenté sur la figure 6 ci-dessus, le solveur reçoit en paramètre des variables (V_1, \dots, V_p) qui ont chacune leur propre domaine (D_1, \dots, D_p) et les contraintes qui peuvent subsister entre chacune d'elles. Puis le solveur résout son problème en local et donne dans le meilleur des cas une solution. Dans le cas où il n'y a pas de solutions, il renvoie que le problème est insatisfiable.

Ainsi pour résoudre notre problème défini sur la figure 1, on passera en paramètre :

- les variables : {pare-chocs, capote, caisse, enjoliveurs}
- le domaine de chaque variable :
 $D(\text{pare-chocs})=D(\text{capote})=D(\text{caisse})=D(\text{enjoliveurs})= \{\text{jaune, bleu, rouge}\}$
- les contraintes : {(enjoliveurs=rouge, caisse=rouge),
(enjoliveurs=jaunes, caisse=jaune), (caisse=bleu,pare-chocs=bleu),
(caisse=rouge,pare-chocs=rouge), (caisse=jaune,pare-chocs=jaune),
(pare-chocs=bleu, capote=bleu), (pare-chocs=rouge, capote=rouge),
(pare-chocs=jaune, capote=jaune), (caisse=bleu, capote=bleu),
(caisse=jaune, capote=rouge), (caisse=rouge, capote=jaune)}

Et le solveur nous retournera comme solution : **INSATISFIABLE**

2.2. Résolution d'un problème distribué

Un réseau de contraintes distribué est un réseau de contraintes qui est naturellement réparti entre plusieurs agents sur différentes machines, et pour lequel il est impossible ou peu souhaitable de réunir l'information sur un même site pour résoudre le problème par les méthodes classiques. Formellement, c'est un quintuplet (X, D, C, A, φ) où X , D et C sont les variables, domaines et contraintes du problème global, A est l'ensemble des agents, et $\varphi: X \rightarrow A$ est la fonction qui fait correspondre à chaque variable un agent. Dans ce contexte, on considère généralement que chaque agent possède en plus de ses variables, les contraintes qui impliquent au moins l'une d'entre elles.

La résolution d'un problème en distribuée se fait à l'aide d'un ensemble de machines et d'agents.

On suppose que l'utilisateur souhaitant résoudre son problème en distribué se trouve sur la machine M_1 . Il va alors saisir son problème sur cette machine et lancer la résolution. Chaque partie de problème va être distribuée aux agents (A_y) qui constituent la plateforme, sachant qu'il peut y avoir plusieurs agents par machine. Chaque agent va résoudre son problème local (en faisant appel au solveur) puis communiquera avec les autres agents pour échanger des messages (par exemple lorsqu'un agent aura trouvé une solution, il la communiquera à un voisin pour que celui-ci résolve son problème en tenant compte de cette solution). Pour terminer, une fois que le problème aura été résolu, une solution sera indiquée à l'utilisateur. A noter que tout au long de la résolution du problème, l'utilisateur peut recevoir des informations telles le temps de calcul, le nombre de messages échangés, ...

Tout ceci pourrait se représenter par la figure 7 dessinée ci-dessous :

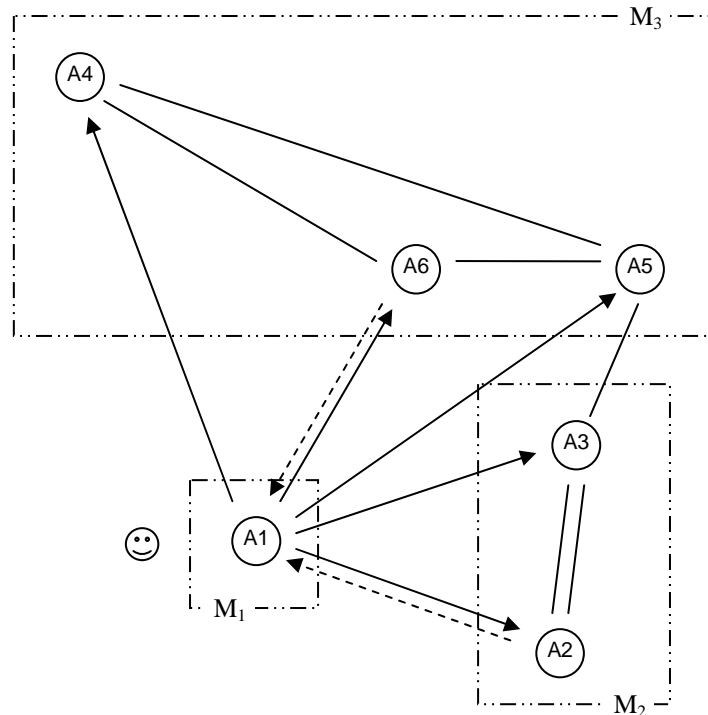


Fig. 7 – Un problème CSP distribué

— Légende —

☺	Un utilisateur	→	Envoi d'un problème
M _x	Une machine	- - ->	Envoi de résultats (temps de calculs, ...)
A _y	Un agent	—	Echange de données

Par conséquent, le problème définit dans la partie « 2.1.1 Résolution d'un problème centralisé » pourrait se distribuer en incluant les quatre variables (enjoliveurs, caisse, pare-chocs, capote) dans quatre agents distincts, pouvant se trouver sur des machines différentes. Ainsi, quatre des sept agents de la figure 7 seraient définis.

Pour conclure, il est nécessaire de préciser que la résolution d'un problème en distribué n'est pas plus rapide que la résolution d'un problème en local, mais bien plus lente. De ce fait, la distribution d'un problème n'est pas un choix mais une difficulté supplémentaire.

3. Recherche

3.1. Cahier des charges

3.1.1. Objectifs

L'objectif de notre projet est d'implémenter une plateforme permettant de tester différents algorithmes de résolution de CSP distribués dans des conditions réellement distribuées. En effet, une telle plateforme n'existant pas, chaque nouvel algorithme est testé indépendamment des autres, certains parfois même en conditions centralisées ; ce qui paraît plutôt gênant pour des algorithmes dit "distribués".

Dans un premier temps, il nous est demandé d'étudier différents langages et interfaces (en C: socket, MPI, PVM... en Java : MadKit...) afin de déterminer le plus adéquat (en terme de facilité d'utilisation pour l'utilisateur, de performance, de load-balancing...).

D'autre part, il faudra nous familiariser avec les différents algorithmes existants afin d'en extraire les points communs pour déterminer avec précision les besoins de la plateforme.

Enfin, il nous est demandé de programmer un panel de ces algorithmes dans le double but de procéder à la validation de la plateforme et de réaliser une série d'expérimentations.

3.1.2. Niveaux d'utilisations

3.1.2.1. Simple utilisateur

L'idée de base de notre plateforme est de résoudre des problèmes de satisfaction de contraintes en utilisant des algorithmes distribués. Ainsi, l'utilisateur saisira son problème, définira les machines qui devront être utilisées par la plateforme, puis sélectionnera l'algorithme de résolution avant de lancer la résolution. Un laps de temps après, il obtiendra la/les réponse(s) à son problème, ou bien même un message lui signalant qu'il n'y a pas de solutions.

3.1.2.2. Chercheur

Un deuxième niveau de spécification permet de tester ses propres algorithmes distribués. Ainsi le chercheur réutilisera toutes les structures et les classes mises à sa disposition dans la plateforme. Une fois son algorithme relié à la plateforme, il pourra récupérer un ensemble d'informations, dont le temps de calcul ou le nombre de messages échangés entre les agents, et voir ainsi la qualité de son algorithme.

3.2. Outils

3.2.1. Matériel

Pour réaliser notre projet, nous avons utilisé les ordinateurs mis à notre disposition en libre service par l'université Montpellier II. Ceux-ci sont des machines équipées de processeurs Intel Pentium III ou IV avec 256Mo de mémoire vive. Ces ordinateurs ont pour système d'exploitation un système Linux RedHat 7.2. Pour le développement de la plateforme, nous avons utilisé un éditeur de texte installé en standard.

3.2.2. Langage de programmation

Nous avons réfléchi sur le langage de programmation nécessaire à la réalisation de notre projet, sachant que nos connaissances nous permettaient de développer plus rapidement en Java ou en C++. Ceci étant, dans un premier temps, nous avons bloqué notre choix sur C++ uniquement pour sa rapidité d'exécution par rapport à Java. Toutefois, le but d'une plateforme distribuée est de pouvoir comparer différentes méthodes. Ainsi, que le langage ne soit pas le plus rapide importe peu, puisque à partir du moment où l'ensemble de la plateforme est implémenté de la même façon, le facteur temps reste le même. Finalement, comme les deux langages étaient utilisables pour le développement de cette plateforme, nous avons décidé de choisir le langage C++ qui nous semblait le plus opportun et avec lequel nous avons le plus de facilité.

3.2.3. Interface de communication entre chaque processus.

Afin de dialoguer, d'échanger des messages entre plusieurs processus qui peuvent se trouver sur plusieurs machines distinctes, il nous est nécessaire d'utiliser une interface de communication. Pour cela, nous avons deux possibilités qui sont soit d'utiliser la librairie MPI soit la librairie PVM. Une autre solution possible serait de définir nous même le dialogue entre chaque processus à l'aide de la classe Socket ou bien en utilisant MadKit.

3.2.3.1. Madkit

MadKit est une plateforme Java qui construit un ensemble d'agents. MadKit offre les principaux outils pour gérer les agents (gestion, transmission des messages, distribution, ...). Elle permet de diversifier l'architecture des agents, le mode de communication. Etant donné que nous n'avons pas choisi le langage Java, cette plateforme a été écartée. Il est vrai que nous aurions pu interfacier du Java avec du C++ mais cela aurait alourdi notre plateforme alors qu'il existe déjà des outils en C++.

3.2.3.2.PVM

PVM (Parallel Virtual Machine) est une collection de machines réelles sur un même réseau sur lesquelles s'exécute un démon/processus qui contrôle les processus utilisateurs au sein d'une application PVM et qui coordonne les communications entre les machines qui composent la machine PVM. Un démon est présent sur chaque ordinateur et pour chaque utilisateur de la machine PVM et maintient la configuration globale et locale de la machine PVM.

La librairie PVM, qui fonctionne avec C/C++, permet de communiquer avec ces démons pour créer et détruire des processus, gérer des tampons de communication, envoyer, recevoir (point à point ou par broadcast) des messages (de façon bloquante ou non bloquante), synchroniser des processus (par barrières de synchronisation) connaître l'état de la configuration de la machine PVM et pour la modifier (dynamiquement)...

La librairie PVM est intéressante quand les applications tournent sur des réseaux hétérogènes, lorsqu'il existe une bonne interopérabilité entre les différents hôtes. PVM permet le développement d'applications tolérantes aux fautes qui peuvent apparaître lorsqu'un hôte ou une tâche échoue. D'autre part, parce que le modèle PVM est construit autour du concept de machine virtuelle, il met à disposition un puissant jeu de gestion de ressources dynamiques et génère des fonctions de contrôles. Toutefois l'utilisation de cette API semble trop complexe pour le travail que nous voulons réaliser.

3.2.3.3.MPI

MPI (Message Passing Interface) est une librairie pour échanger des messages. On peut l'utiliser avec les langages C et Java. MPI-2 peut s'utiliser en plus avec le langage C++. MPI est connue pour être rapide dans un système multiprocesseur.

MPI a beaucoup plus d'options que PVM tant pour la communication point à point que pour la communication collective. Cette librairie est favorisée lorsqu'une application est structurée pour exploiter des modes spéciaux de communication non disponibles dans PVM. Cela peut être important si un algorithme dépend de l'existence d'une option spéciale de communication. Bien que cette librairie pourrait convenir de part ses fonctionnalités, un très gros travail d'apprentissage est nécessaire et son utilisation complexe, notamment son installation, risquerait de nous prendre beaucoup de temps.

3.2.3.4.Socket

Au final, nous avons choisis d'utiliser l'interface TCP/IP via l'interface Socket de Unix. Ainsi il nous faudra programmer nous même un module de communication entre les différents agents et machines sachant que le principal avantage résidera dans le fait que nous ne dépendons d'aucune librairie. Nous programmerons uniquement les fonctions nécessaires à notre plateforme et maîtriserons parfaitement le comportement de celles-ci. En revanche, l'inconvénient sera une charge de travail plus importante dans le développement ; cela reste à nuancer avec le fait que nous n'aurons pas à apprendre précisément le fonctionnement d'une librairie de communication.

Le choix du protocole de communication TCP (Transport Communication Protocol) se justifie par sa fiabilité (prise en charge d'accusé réception) par rapport au protocole UDP (User Datagram Protocol). Il a aussi l'avantage de garantir l'ordre d'arrivée des paquets transmis et de ne pas dupliquer ceux-ci.

3.2.4. Générateur d'instances

Pour tester notre plateforme, il nous faut générer un problème. Ainsi, nos encadrants nous ont mis à disposition un générateur d'instances qui va générer un ou plusieurs fichiers contenant un problème quelconque.

3.2.4.1.Utilisation du générateur

Un problème peut être généré de deux façons différentes:

- soit nous générons autant de fichiers que d'agents souhaités, où chaque fichier contient uniquement les informations relatives à un agent, c'est-à-dire ses variables et les contraintes inter et intra agents qui se rapportent à lui. De ce fait, nous avons un point de vue local sur le problème lorsqu'un fichier est édité.

- soit nous générons un unique fichier qui contient toutes les contraintes inter agents et intra agents. Ainsi, par l'intermédiaire de ce fichier, nous avons un point de vue global sur le problème. Cependant, l'inconvénient de cette génération est qu'il nous faut par la suite découper nous-même le problème pour le répartir sur plusieurs agents. L'avantage est que l'on répartit le problème à notre façon, c'est pourquoi nous avons choisit cette méthode pour le test de notre plateforme.

Pour générer un problème, nous utilisons l'exécutable nommé 'disgenerator' en lui passant un ensemble de paramètres comme dans l'exemple suivant :

```
./disgenerator 9 5 6 6 3 50 8 3 1 7326 > probleme.csp
```

Une fois exécuté, ce programme stocke tous les paramètres dans l'entête du fichier, afin de mieux le comprendre lorsque celui-ci est édité.

Chaque paramètre de gauche à droite a une signification bien précise :

- '9' correspond au nombre total de variables du problème. Celles-ci seront notées dans ce cas de v_0 à v_8 .
- '5' indique la taille du domaine des variables. Ici chaque variable aura un domaine de 5 valeurs allant de 0 à 4.
- '6' spécifie le nombre de contraintes inter agents du problème. Une contrainte est représentée par un nom, le nom des variables concernées par la contrainte et une liste de couple de valeurs interdites.
- '6' représente la densité des contraintes du problème, plus précisément le nombre de couples de valeurs interdites. Ici, une densité de 6 signifie que chaque contrainte inter agent aura 6 couples de valeurs interdites.

```
c0 nogood v1 v5
0 2
3 4
3 2
2 0
1 2
1 4
```

Par exemple la deuxième ligne signifie que si la variable 'v1' prend la valeur '0', alors la variable 'v5' ne peut pas prendre la valeur '2', et inversement. Les autres lignes se lisent de la même façon.

- '3' est le nombre total d'agents. Par conséquent, comme nous avons 9 variables et 3 agents, chaque agent reçoit 3 variables de la façon suivante:
 - L'agent '0' aura les variables 'v0' à 'v2',
 - L'agent '1' aura les variables 'v3' à 'v5',
 - L'agent '2' aura les variables 'v6' à 'v8'.
- '50' est un pourcentage qui représente le nombre de contraintes intra agents que l'on veut créer à partir du nombre de contraintes inter agents. Comme nous avons 6 contraintes inter agents, avec ce pourcentage nous obtenons 3 contraintes intra agents.
- '8' est aussi un pourcentage qui représente le nombre de couples interdits qui seront attribués aux contraintes intra agents. Ici, chaque variable a un domaine de 5, et comme une contrainte porte sur 2 variables, la densité maximale des contraintes intra agents peut être de $5*5=25$. Et comme nous ne voulons que 8% de la densité maximale, nous obtenons 2 couples de valeurs interdites. L'exemple ci-dessous montre que la contrainte intra agents porte sur les variables 'v0' et 'v1' qui sont 2 variables de l'agent 0.

```
c6 nogood v0 v1
2 4
4 0
```

- '3' est un paramètre qui permet d'indiquer le point de vue choisit pour la génération de ce fichier. Comme nous voulons le point de vue global, il faut mettre un nombre égal au nombre d'agents, c'est-à-dire 3.

Pour générer un point de vue local à un agent, il aurait fallu mettre le numéro de l'agent que l'on désirait, c'est-à-dire de 0 à 2.

- '1' est le nombre d'instances que l'on veut générer.

- '7326' représente la racine de la génération aléatoire. Ce nombre est utile lorsque nous générons le point de vue de chaque agent. Il permet de re-générer le même problème, mais vu d'un agent différent.

3.2.4.2.Exemple d'un fichier généré

Sur l'exemple ci-dessous, nous avons un fichier qui contient notre problème. Celui-ci utilise 3 agents qui sont notés de 0 à 2 inclus.

En fait, en regardant plus précisément, on s'aperçoit que ce fichier représente le point de vue de l'agent 3 qui voit l'ensemble du problème.

```
Agent 3 has variables 0 to 8
3 agents
9 variables
5 valeurs par variable
6 contraintes inter
6 densite des contraintes
inter
50 % contraintes intra
8 % densite des contraintes
intra
1 num instance
#
v0 0 1 2 3 4 *
v1 0 1 2 3 4 *
v2 0 1 2 3 4 *
v3 0 1 2 3 4 *
v4 0 1 2 3 4 *
v5 0 1 2 3 4 *
v6 0 1 2 3 4 *
v7 0 1 2 3 4 *
v8 0 1 2 3 4 *
#
c0 nogood v1 v5
0 2
3 4
3 2
2 0
1 2
1 4
*
c1 nogood v0 v4
1 1
3 1
2 3
3 2
2 4
1 2
*
c2 nogood v4 v7
3 4
4 3
1 1
2 2
1 4
2 1
*
c3 nogood v0 v6
3 2
4 2
0 1
0 0
1 2
1 0
*
c4 nogood v0 v5
3 3
0 4
3 1
0 0
1 1
2 0
*
c5 nogood v1 v8
4 3
3 2
3 1
4 4
2 2
3 4
*
c6 nogood v0 v1
2 4
4 0
*
c7 nogood v3 v4
2 4
3 1
*
c8 nogood v7 v8
1 4
4 1
*
#
@
.
```

3.2.5. Solveur

3.2.5.1. Possibilités

Dans un premier, nous devons trouver un solveur déjà implémenté et qui peut être réutilisable, c'est à dire que nous puissions l'inclure dans notre plateforme pour résoudre un problème en local. Par la suite, il sera possible d'implémenter soit même un solveur local et de l'incorporer dans notre plateforme.

D'après la liste fournie en [WI], nous avons effectué un tri des solveurs en supprimant tout d'abord ceux qui n'étaient pas en C++ puis en éliminant ceux qui nous paraissaient complexe. Nous avons extrait quelques solveurs possibles, expliqués brièvement ci-dessous.

- **Choco** est un petit système qui met à notre disposition des outils basiques comme des algorithmes et des structures de données nécessaires dans à n'importe quel système de contraintes. Ce module s'utilise avec le langage de programmation Claire, qui est très peu répandu.
- **EFC** est un solveur de CSP. Il est assez souple et il a déjà été utilisé pour implémenter plus de 50 algorithmes différents de backtracking. Il utilise aussi les mécanismes de NoGood
- **Peter van Beek's C library** est une bibliothèque étendue de routines pour l'expérimentation de différentes méthodes de backtracking qui permet de résoudre les CSP binaire.
- **Tudor Hulubei's C++ CSP Library** est un ensemble de classes qui peuvent être utilisées pour représenter et résoudre les CSPs. Cette librairie n'est pas un solveur général. En effet, elle ne supporte pas de nombreux dispositifs qui sont utiles pour représenter les problèmes de la vie réelle comme les domaines continus, les contraintes n-aires, ... Elle est cependant utile pour expérimenter de nouveaux algorithmes, réaliser des tests de performances, ...

3.2.5.2.Essais

Nous avons dans un premier temps tenté d'utiliser EFC. Cependant la robustesse d'installation et l'incompatibilité de certains outils mis à notre disposition par l'université (compilateur g++ ou tout simplement pour des raisons de droits d'accès), ne nous ont pas permis son utilisation. Et pour les mêmes raisons, nous n'avons pu utiliser les deux derniers solveurs cités ci-dessus.

Ainsi dans un second temps, nous nous sommes orientés vers le solveur Choco qui est une extension du module Claire. Nous avons appris à le faire fonctionner et avons pu résoudre des problèmes simples. Mais malheureusement, en poussant un peu plus dans notre apprentissage, nous nous sommes aperçu qu'il nous était impossible de l'utiliser car nous n'avons trouvé aucun moyen pour l'intégrer à notre plateforme puisque Choco est un langage interprété. En effet, pour lancer un problème, il nous fallait être dans le prompt du module Choco et il était nécessaire de recompiler le problème à chaque fois.

Par conséquent, ne trouvant pas de solveurs appropriés, nos encadrants nous ont proposé d'utiliser un solveur écrit par un chercheur du Lirmm, à savoir Christian Bessière.

3.2.5.3.Solveur choisit

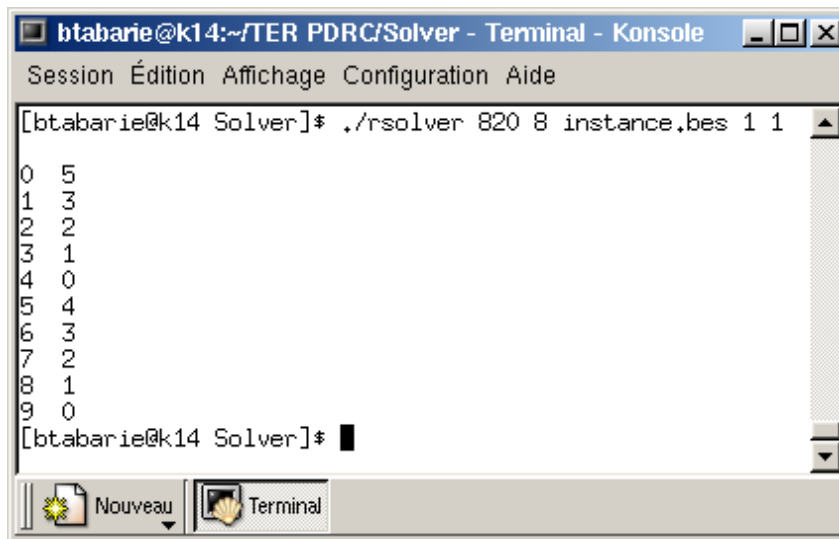
Nous avons donc choisit le solveur de Christian Bessière qu'il nous faudra par la suite intégrer à notre plateforme.

Pour que le solveur résolve notre problème, nous devons notamment lui passer en paramètres :

- L'algorithme de résolution à utiliser, qui est à choisir parmi une liste de 27 algorithmes.
- Le problème, qui se situe dans un fichier dont le contenu est valide et bien formé afin qu'il soit compréhensible par le solveur.

Lorsque nous exécutons le solveur avec les paramètres adéquats, celui-ci nous retourne une solution possible. Dans le cas où il n'y a pas de solutions, le solveur nous retourne un code d'erreur pour nous signaler que le problème est insatisfiable.

Voici un exemple lorsque le problème est satisfiable :



```
btabarie@k14:~/TER_PDRC/Solver - Terminal - Konsole
Session Édition Affichage Configuration Aide
[btabarie@k14 Solver]$ ./rsolver 820 8 instance.bes 1 1
0 5
1 3
2 2
3 1
4 0
5 4
6 3
7 2
8 1
9 0
[btabarie@k14 Solver]$
```

Le solveur est exécuté avec un ensemble de paramètres qui sont :

- '820' pour le code de l'algorithme nommé AC2001
- 'instance.bes' pour indiquer le fichier contenant notre problème
- '8', '1' et '1' sont des paramètres particuliers qui nous importent peu

La résolution s'est bien déroulée, par conséquent le solveur nous retourne une solution qui se lit de la façon suivante :

- la variable 0 prend la valeur 5,
- la variable 1 prend la valeur 3,
- la variable 2 prend la valeur 2,
- ...
- la variable 9 prend la valeur 0.

4. Développement

4.1. Architecture générale de la plateforme

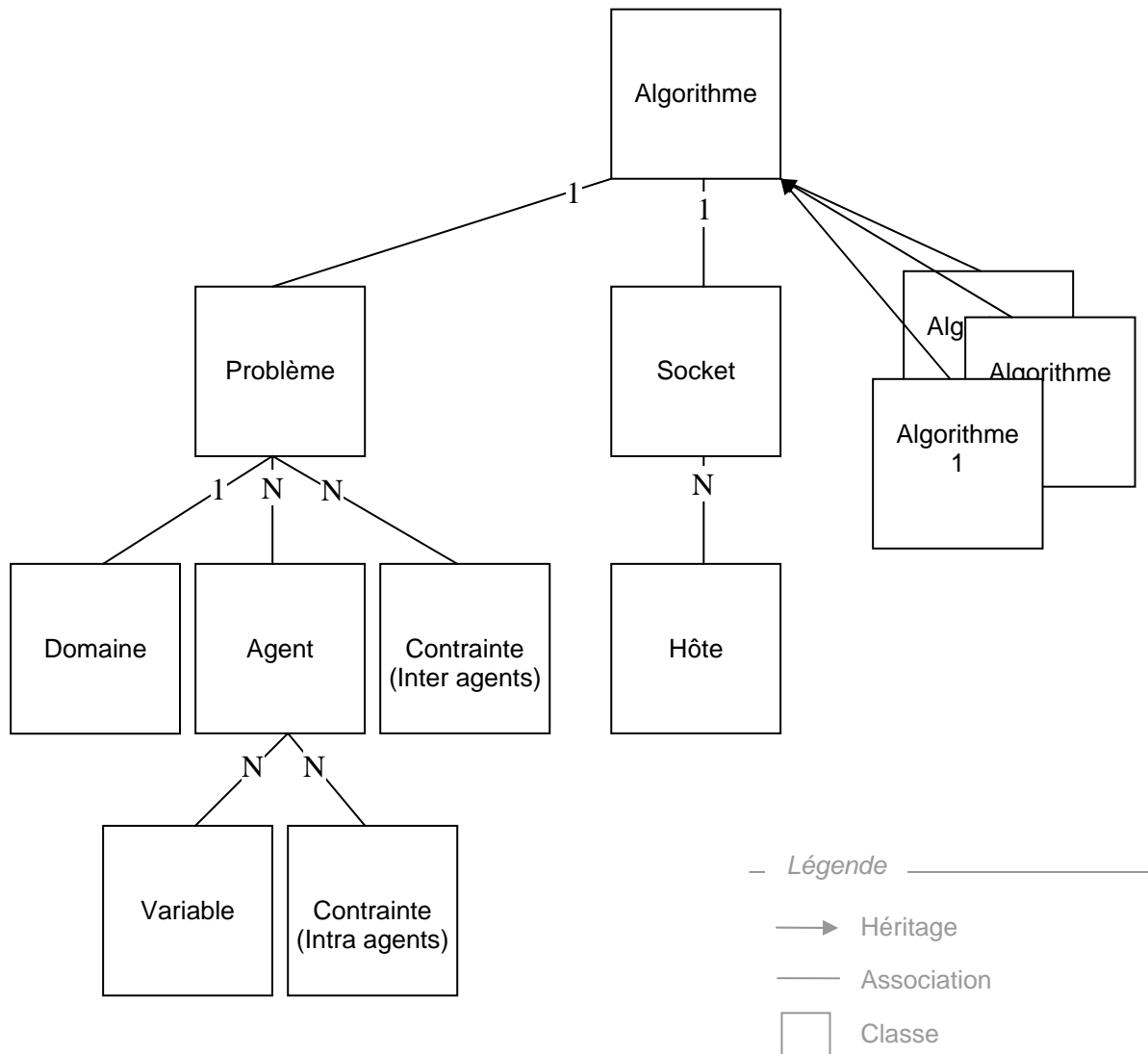


Fig. 8 – Schéma général de notre application

Comme définit dans la figure 8, nous avons défini une classe *Algorithme* qui va contenir le problème et l'interface de communication *Socket*. Cette classe *Algorithme* devra être redéfinie (modélisée par l'héritage sur le schéma ci-dessus) par le chercheur en utilisant la méthode `void Algorithme::start(int)` où l'entier qui lui est passé en paramètre représente le numéro du processus dans lequel est lancé l'algorithme.

La classe *Problème* contient le domaine initial (qui contient les valeurs possibles du domaine pour chaque variable), plusieurs agents et plusieurs contraintes. Ces contraintes représentent les contraintes inter agent.

D'autre part, un agent contient plusieurs variables et plusieurs contraintes, qui sont les contraintes intra agents. La classe *Variable* contient un entier identifiant la variable elle-même et une chaîne de caractère contenant son nom d'origine. La classe *Contrainte* contient 2 entiers $v1$ et $v2$ référençant des variables et des couples de valeurs interdites du domaine.

L'autre classe qui est utilisée par la classe *Algorithme* est la classe *Socket* qui contient tous les moyens de communication entre les divers processus. Elle contient une liste d'hôtes qui référencent toutes les connexions possibles vers les autres processus.

Pour fonctionner, cette classe lance un thread (processus léger) en écoute de connexion sur un port TCP. A chaque connexion, un nouveau thread en écoute de message est lancé. A chaque arrivée de messages, la fonction `bool Algorithme::message_recu(Message)` de la classe *Algorithme* est appelée par le thread. Si cette fonction (qui peut être redéfinie par le chercheur) renvoie vraie, le message '*Message*' est mis à la suite dans une liste, sinon il est rejeté. Ainsi, nous avons défini une fonction qui permet soit de récupérer les messages '*Message*' soit d'en attendre un s'il n'y en a pas.

Enfin, nous avons défini une classe *Message* que nous n'avons pas inclus dans la figure 8 car celle-ci est utilisée par notre plateforme pour représenter une information qui transite d'un processus à un autre. Ainsi, elle est indépendante de notre plateforme. Nous laissons la possibilité aux chercheurs de définir leur propre type de messages en respectant le format des entiers (Little Endian ou Big Endian).

4.2. Construction du problème

Le problème est construit par l'intermédiaire de la classe *CspFile* qui analyse le fichier *csp*.

Durant toute la phase de construction du problème, nous accéderons toujours à l'instance de l'objet '*Probleme*', créée dans la classe *Algorithme*, grâce à la méthode *get_probleme()*.

Après analyse du fichier, plusieurs étapes sont nécessaires pour construire le problème. Tout d'abord, nous définissons la taille du problème grâce à l'appel '*get_probleme().set_size(<nombre d'agents>, <nombre de contraintes inter agents>)*'. Puis nous définissons le nombre de variables et le domaine du problème à l'aide de l'appel '*get_probleme().set_domaine().set_size(<nombre de variables>, <nombre de valeurs dans le domaine>)*'. Ensuite, nous définissons le nombre de variables pour chaque agent grâce à '*get_probleme().set_agent(<numéro de l'agent>).set_variable_size(<nombre de variables de l'agent>)*'. Après cela, pour chaque variable, nous définissons son nom ainsi que les valeurs de son domaine à l'aide de '*get_probleme().add_variable(<nom de la variable>)*' et '*get_probleme().add_domaine(<valeur à ajouter au domaine de cette variable>)*'. Enfin, pour terminer, chaque agent reçoit ses contraintes inter et intra agents grâce aux méthodes qui définissent :

- le nombre de contraintes intra (ou inter) agents avec '*get_probleme().set_agent(<numéro de l'agent concerné>).set_contrainte_size(<nombre de contraintes intra ou inter agents>)*'
- pour chaque contrainte :
 - le nom des variables concernées par '*get_probleme().set_agent(<numéro de l'agent concerné>).set_contrainte(<nom de la contrainte >).set_v1(<nom de la variable 1>)*' et '*get_probleme().set_agent(<numéro de l'agent concerné>).set_contrainte(<nom de la contrainte>).set_v2(<nom de la variable 2>)*'
 - la densité par '*get_probleme().set_agent(<numéro de l'agent concerné>).set_contrainte(<nom de la contrainte >).set_size(<densité de la contrainte>)*'
 - les couples de valeurs interdites par '*get_probleme().set_contrainte(<nom de la contrainte>).set_d1(<valeur de la variable 1>)*' et '*get_probleme().set_contrainte(<nom de la contrainte>).set_d2(<valeur de la variable 2>)*'

4.3. Incorporation du solveur dans la plateforme

Nous avons créée une classe qui va permettre la résolution d'un problème à l'aide du solveur relié à la plateforme, qui est actuellement celui de Christian Bessière. Nous utilisons le terme « relié » et non pas « intégré » car nous n'avons pas pu avoir accès au source du solveur car il n'est pas open source. Cependant, ceci peut être un avantage puisqu'il sera possible par la suite d'intégrer un autre solveur compilé puisque nous nous servons de fichiers et faisons un appel au solveur déjà compilé.

Cette classe s'utilise en lui passant en paramètre un problème qui contient tous les agents, toutes les variables et toutes les contraintes inter et intra agent. Puis pour utiliser ce solveur, il nous faut l'interfacier.

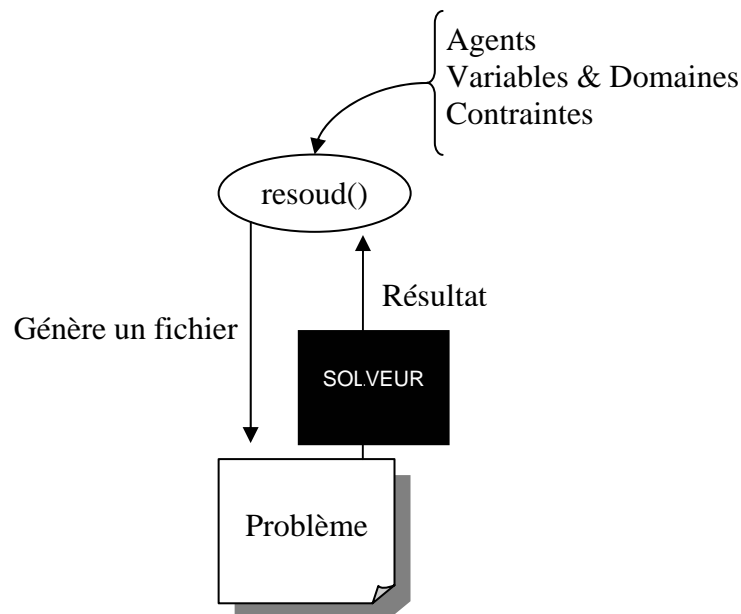


Fig. 9 - Utilisation du solveur par la plateforme

Ainsi notre stratégie, comme défini sur la figure 9 ci-dessus, consiste à créer un fichier utilisable par le solveur, à exécuter le solveur et enfin à récupérer le résultat. Pour cela nous avons défini une méthode dans cette classe qui va avoir comme paramètre les variables et leurs domaines de chaque agent ainsi que les contraintes intra agent et les contraintes externes. Cette méthode appellera le solveur et en résultat, elle retournera un booléen, dont la valeur est vraie si elle réussit à obtenir une solution au problème posé, faux sinon, et génère une instance contenant le résultat que nous récupérons.

5. Résultats

Les résultats seront fournis ultérieurement en annexe

6. Discussion

Dans cette partie, nous allons avoir un regard critique sur le travail effectué au cours de notre TER, des contraintes matérielles en passant par les contraintes logicielles jusqu'au résultat final.

La première contrainte que nous avons eue fut de trouver un système pour l'échange de données entre chaque agent. De ce fait nous nous sommes aperçu, que de par notre étude de plusieurs librairies, qu'il était plus judicieux de développer nous même le mode de communication intra agent plutôt que de dépendre d'une quelconque librairie. Ainsi nous espérions donner à notre plateforme une réelle autonomie en n'ayant aucun lien avec des éléments extérieurs dont nous aurions été tributaire. Malheureusement, cette indépendance va disparaître comme cela est expliqué dans le paragraphe suivant.

La deuxième contrainte que nous avons eue a été de trouver un solveur utilisable dans la plateforme. Si de nombreux solveurs existent, beaucoup de part leurs caractéristiques diffèrent, et comme il n'existe pas de solveurs universels, la tâche fut d'autant plus dure dans la prospection, en gardant à l'esprit que les inconvénients ne devaient que peu influencer sur le résultat souhaité. Ainsi, nous nous sommes orientés dans un premier temps vers un solveur développé en C++ en espérant pouvoir l'inclure facilement dans la plateforme. Mais le matériel mis à notre disposition ne nous a pas permis de tester autant de solveurs que nous l'aurions souhaité. En rien nous ne critiquons ce matériel, mis à notre disposition pour nos études, car nous comprenons que pour des raisons de sécurité et de fonctionnement optimal, certains droits ou caractéristiques ne pouvaient être mis en place. Subséquemment, les quelques solveurs que nous avons pu tester ne nous ont pas permis d'aboutir à un résultat convenable pour notre plateforme et c'est donc pour cela que nos encadrants nous ont mis à notre disposition le solveur de M. Christian Bessière. Ce solveur n'étant pas libre de droits, nous avons trouvé un système permettant d'utiliser le solveur compilé sans toucher au code source. Cependant cela constitue un avantage puisque nous rendons notre plateforme dépendante d'un quelconque solveur. Ainsi, il sera possible pour un chercheur de développer son propre solveur, en respectant quelques spécifications pour son intégration dans la plateforme, et de l'utiliser avec celle-ci.

7. Conclusion

Nous avons réalisé une plateforme distribuée pour les réseaux de contraintes dans le cadre notre projet de Travail d'Etude et de Recherche de maîtrise informatique. Cette plateforme permet de résoudre des problèmes de satisfaction de contraintes en les distribuant réellement. Nous avons donné la possibilité à un chercheur d'inclure un algorithme quelconque développé par ses propres soins afin que celui-ci puisse le tester et ainsi s'apercevoir de sa qualité. Lorsqu'un utilisateur souhaite uniquement résoudre un problème, il a la possibilité de choisir parmi plusieurs algorithmes de résolution qui viendront petit à petit étoffer la plateforme lorsque un chercheur aura jugé utile de les y inclure.

Toutefois, afin d'améliorer la plateforme, nous aurions pu développer notre propre solveur qui aurait permis à notre plateforme d'être totalement indépendante. Cependant, le temps et la complexité de cette tâche ont été les principaux facteurs qui nous ont rebuté. Cette partie pourra peut être faire l'objet d'un autre projet.

Ce projet nous a permis d'appliquer les connaissances qui nous ont été inculquées au cours de ces deux années de Licence et de Maîtrise Informatique à l'université des sciences Montpellier II ainsi que de nous initier à la recherche dans une optique éventuelle de poursuite d'études dans ce domaine. Nous avons été confrontés à de nombreux problèmes et dans la plupart des cas nous avons pu trouver une solution alternative afin de les résoudre partiellement. Enfin ce projet aura été l'occasion de découvrir et d'utiliser des outils dont nous n'avions pas la moindre idée de leurs existences.

Références

[W0] www.irisa.fr/iHPerf2000/transparentes/Desprez.ppt

[W1] <http://www.4c.ucc.ie/web/archive/solver.jsp>

[W2] <http://www.inra.fr/bia/T/schiex/Export/HDR.pdf>

[W3] <http://casteyde.christian.free.fr/cpp/cours/>

[W4] <http://www.choco-constraints.net/>

[W5] <http://www.google.fr/>

[W6] <http://www710.univ-lyon1.fr/~csolnon/Site-PPC/e-miage-ppc-som.htm>

[W7] <http://www.sciences.univ-nantes.fr/info/perso/permanents/goualard/Teaching/Maitrise/UOM3/cours-2-uom3-2002.pdf>

-- Résumé --

Dans le cadre du Travail d'Etude et de Recherche de Maîtrise Informatique, nous avons développé une plateforme distribuée pour les réseaux de contraintes. Cette plateforme, implémentée en C++, permet de résoudre des problèmes de satisfaction de contraintes en les distribuant sur un ensemble de machines. L'autre finalité de cette plateforme est la possibilité d'y inclure un ensemble d'algorithmes afin de les tester, tant au niveau de la rapidité que de l'efficacité. Le fait de développer une telle plateforme s'est avéré utile car dans la plupart des cas, des algorithmes distribués étaient utilisés en local pour résoudre des problèmes. Ainsi, par le biais de cette plateforme, nous donnons la possibilité de résoudre des problèmes de satisfaction de contraintes dans des conditions réellement distribuées.

Nous avons dans un premier temps étudié un ensemble d'outils nécessaire au développement tel des bibliothèques pour l'échange de messages mais nous nous sommes aperçu qu'il était plus judicieux de développer nous même cet échange. Puis nous avons implémenté un algorithme pour tester notre plateforme.