

# Proyecto Inteligencia Artificial

## Entrega 2

Nicolás Troncoso Carrère  
9973060-9  
email: ntroncos@inf.utfsm.cl

25 de mayo de 2004

### **Resumen**

En este documento se detallan los resultados de un algoritmo completo usando FC+GBJ para resolver CSP. Se presentan comparativas de los resultados obtenidos con el algoritmo del autor. Adicionalmente se hace un poco de historia y descripción del problema.

## **1. Estado del Arte**

El problema de satisfacción de restricciones nació en el estudio de inteligencia artificial y la computación gráfica hacia los años 60. Existe un algoritmo fácil para resolver este problema: enumeración exhaustiva, el problema es que esto puede tardar un largo tiempo ya que al agregar una restricción el tamaño del problema aumenta de forma exponencial. Hoy en día es un problema que aparece en todas las ramas de la ciencia y negocios bajo distintas formas. De aquí ha nacido un gran interés por resolver este problema de forma eficiente (tiempo razonable). Distintos enfoques se han adoptado para tratar este problema, los más eficientes ha sido los algoritmos incompletos. Estos buscan una solución aproximada a la mejor de las soluciones (óptimo), los últimos algoritmos utilizados son los algoritmos evolucionistas.

## 1.1. Nueva Representación

El CSP siempre ha padecido del problema de revisar las restricciones, y para ello se debe ir verificando los vecinos. Estas verificaciones tienen gran impacto en el desempeño de los algoritmos por lo que *I. Juho* and *A. Tóth* and *J.I. van Hemert* crearon un nuevo enfoque para poder minimizar este limitante de desempeño. Lo que hacen es agrupar elementos con el mismo dominio (color) en **Súper Nodos** y unirlos con otros nodos o súper nodos con **Súper Arcos**. Esta nueva representación mejora el desempeño de la verificación de restricciones. Para ser probado se implementó en un algoritmo incompleto y se obtuvieron resultados prometedores[4].

## 2. Modelo

Formulación del modelo general para la resolución de CSP. Se define el problema como una tupla  $(X, D, C)$ , donde  $X$  es un conjunto finito de variables,  $D$  es una función que mapea cada variable  $X_i$  a un conjunto de valores posibles que puede tomar  $X$ .  $C$  es un conjunto de restricciones descrito por una lista de valores que no pueden tomar las variables del conjunto  $X$  de forma simultánea.

### 2.1. Solución

Una solución se representa como la asociación de una variable  $X$  con un elemento del conjunto  $D$ , para cada variable que pertenece al conjunto  $X$ . A priori no se puede saber si un problema tiene solución, adicionalmente es mucho más caro asegurar que un problema *no* tiene solución, que si tiene.

### 2.2. No Satisfacción de Restricciones

No satisfacción o violación de restricciones ocurren cuando se asigna una de las combinaciones descritas en el conjunto  $C$ . Realizar esta asignación implica un costo que es igual al número de restricciones insatisfechas. Cabe decir que cuando la importancia de las restricciones varía, es decir algunas son más importantes que otras,

el costo asociado a la no satisfacción de cada una varía. El caso estudiado es para restricciones igualmente importantes, es decir, la violación de cualquiera de ellas conlleva el mismo costo.

### 2.3. Modelo de Programación Lineal

Se utiliza un modelo de 4 dimensiones para representar las restricciones, ya que  $C$  está dado por un par de variables y un par de valores que no pueden tomar. Dado esto se utilizan los índices  $i, j$  para las variables y  $k, p$  para los valores respectivos. Suponemos que las restricciones son simétricas.

Función Objetivo:

$$\text{mín} \sum_{i=0}^n \sum_{j=i}^n \sum_{k=0}^m \sum_{p=0}^m (X_{ijkp} \cdot C_{ijkp})$$

Variables:

$$X_{ijkp} \begin{cases} 0 \\ 1 \end{cases}$$

Restricciones:

$$X_{ijkp} \neq 1 \quad j < i \quad \forall k \quad \forall p$$

$$\sum_{i=0}^n \sum_{k=0}^m \sum_{p=0}^m (X_{ijkp}) = 1 \quad \forall j$$

### 2.4. Parámetros del Problema

Los CSP quedan descritos por un conjunto de parámetros que permiten clasificarlos. Estos parámetros pueden ser generados aleatoriamente para obtener problemas de distintos tamaños y complejidades. La definición de los parámetros es la siguiente:

- $n$ : número de variables del problema.
- $m$ : tamaño del dominio (valores que pueden tomar las  $n$  variables).
- $p1 \in [0, 1]$ : medida de conectividad, cantidad de restricciones.
- $p2 \in [0, 1]$ : medida de incompatibilidad.

### 3. Representación

Esta sección describe la forma computacional en la cual sera representado (modelado el problema CSP)

#### 3.1. Entrada de Datos

La entrada de datos será a través de un archivo en texto plano, donde para cada restricción se listan las dos variables involucradas, y todos los pares de valores incompatibles. Adicionalmente se implemento la entrada de datos a través de la STDIN(entrada estándar).

$$X_i \ X_j \ v_{ia} \ v_{ib} \ v_{ic} \ v_{id} \ -1$$

#### 3.2. Salida de Datos

La salida sera a través de un archivo en texto plano, donde se lista cada variable con el valor asignado. Al final del archivo se indica el costo de la solución.

$$X_i \ v_i$$

#### 3.3. Estructura de Datos

Para la representación computacional del problema se utilizan tres estructuras. La primera para la representación del grafo de conexiones. La segunda representa las restricciones de la manera presentada en el archivo. La tercer estructura se compone de las dos anteriores más los dominios de cada variable, y la solución al problema(si existe).

Solo se utiliza memoria dinámica para la representación, se aumenta la complejidad de la implementación pero a cambio se obtiene un menor uso de memoria.

##### 3.3.1. Representación de los Nodos

La siguiente representación nos permite tener el control sobre cuantos nodos adyacentes se tiene, y permite acceso directo al dominio

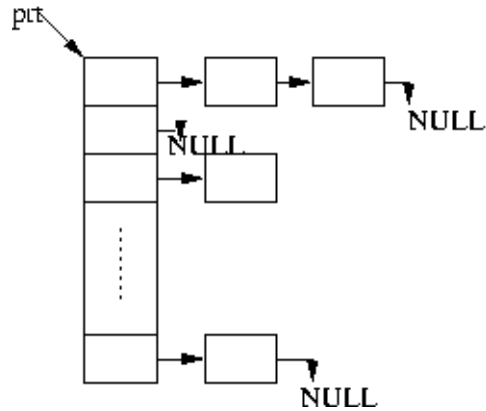


Figura 1: Grafo de Interferencia

del nodo. Para construir el grafo se utiliza una lista de adyacencia (ver Figura 1 pagina 5)

```
typedef struct node{
    int *domain;
    int tag;
    int adyno; /*number of adjacent nodes*/
    struct node *next;
}node;
```

### 3.3.2. Representación de los arcos(restricciones)

La siguiente representación contiene los valores inválidos, así como el número de pares inválidos. Los arcos se representan de la misma manera que los nodos (Figura 1 los unico que cambia es la estructura interior.)

```
typedef struct arc{
    int val1;
    int val2;
    struct arc *next;
    int adyno; /*number of adjacent arcs.*/
}arc;
```

### 3.3.3. Representación del Problema

Esta estructura contiene todo el problema que se intenta resolver. Un grafo de arcos, un grafo de adyacencia, número de nodos, número de restricciones. Adicionalmente contiene un puntero a la solución del problema y un puntero a todos los dominios, estos últimos son referenciados por los nodos también.

```
typedef struct cspdata{
    struct arc    **arcs;
    int arcno;
    struct node  **graph;
    int nodeno;
    int *solution;
    int **domains;
}cspdata;
```

## 4. Pseudocódigo

A continuación se describe el pseudocódigo utilizado para resolver el CSP. El algoritmo es recursivo, y utiliza las heurísticas "mas conectado" "menor dominio". Antes de entrar en la recursión guarda una copia de los dominios y de la solución anterior, y hace crecer una pila con el orden de instanciación de los nodos. Si la recursión falla y se empieza a devolver, se chequea si al punto en que se volvió esta conectado al nodo en conflicto, sino se sigue retornando. Si lo es sigue con el ciclo while de la instancia en a la que haya retornado. Una vez que tiene una solución termina.

```
n:={nodes}
C[n]:={Restricciones}
D[n]:={Domain}
S:={} //Solution
I:={} //instantiation
begin solve_csp()
    if #S equal n then
        return S
    end program
endif
```

```

T:={ }
do
    i:=most conected, least domian NOT IN T
    push(i,T)
    c:=first available value from D[i]
    OLD_D:=D
    OLD_S:=S
    S:=S+(i,c)
    forward_check_domains(D,C,S) /*The new instantiates variable modifies
domains*/
    if some #D is equal to zero then
        return
    endif
    push(i,I)
    solve_csp()
    pop(I)
    r=top(I)
    pop(I)
    if r is not connected to i then
        push(i,I)
        return
    endif
    S:=OLD_S
    D:=OLD_D
while #T distinct n
endbegin

```

## 5. Resultados

No existe data apropiada para hacer un análisis serio de los resultados.

## 6. Conclusiones

El problema de CSP es simple, pero su resolución es compleja. Esta complejidad es mayor aun cuando se trata de hacer una implementación eficiente para su resolución. El algoritmo implementa-

do es un algoritmo completo, es decir busca en todo el espacio de búsqueda de nuestra representación, sin embargo como a mi personalmente solo me importaba encontrar una solución el algoritmo no necesariamente buscaba en todo el espacio de búsqueda. El peor caso es si no encuentra solución, y el caso más optimista es que se instancie cada nodo sin conflictos, por lo que la búsqueda es mínima.

La implementación del programa es la correcta, y permite hacer batch de ejecuciones para hacer pruebas necesarias. De hacerse estas pruebas debieran ser  $n^4p$  pruebas distintas, donde  $n$  es el numero de veces que se varía cada parámetro y  $p$  es el numero de pruebas que se hace para cada conjunto de parámetros, para así obtener una esperanza para la cantidad de chequeos de restricciones.

Los resultados<sup>1</sup> del programa se debieran asemejar a una campana que depende de los parámetros  $p1$  y  $p2$  descritos en la sección 2.4 pagina 3.

Las limitaciones para este tipo de algoritmo esta claramente dado por memoria y tiempo de ejecución; si se elige una implementación simple el techo para la resolución es la memoria, si se opta por una representación más inteligente eficiente en el uso de memoria el techo para la resolución sera el tiempo de ejecución.

Se recomienda el estudio e investigación de algoritmos incompletos para resolver estos problemas.

---

<sup>1</sup>Si desea los resultados contacte al autor

**Referencias:**

[1] <http://kti.ms.mff.cuni.cz/~bartak/constraints/index.html>

Constraint Programing

[2] <http://www.ncst.ernet.in/kbcs/vivek/issues/10.4/ravi/ravi.html>

Constraint Programing Issue 10.4

[3] <http://www.inf.utfsm.cl/~csoza/csp/csp.pdf>

Proyecto de Investigacion Aplicada: CSP

[4] <http://homepages.cwi.nl/~jvhemert/publications-csp.html>

Binary Merge Model Representation of the Graph Colouring Problem