make.tol de Constraint.Action

Programa de ejemplo de solucion de sistemas de restricciones del tipo (restriccion, accion correctora) programados de 3 formas diferentes a) por evaluacion iterativa, b) por recursion pura y c) por 2 funciones recursivas donde la primera se apoya en la segunda. De los 3 metodos es el iterativo el que puede resolver problemas con un mayor numero de pasos, el segundo con mas capacidad es el de 2 funciones recursivas (birecursivo) y el recursivo puro puede producir un desbordamiento de la pila cuando el numero de pasos (acciones y pruebas de restricciones) aumenta. Es un programa desarrollado en un solo fichero Tol en el que se declaran todas las funciones para hechos, restricciones, base de hechos y sus 3 formas de resolucion. En el metodo iterativo, la memoria entre las iteraciones, se implementa con un fichero interno de datos en formato Bst. Es una version basica de programacion con restricciones para uso formativo formativo donde: a) cada restriccion indica su solucion correctora, b) los hechos tienen como valor un solo numero y c) no existe un dominio para los valores de los hechos. Por ello puede resolver algunos problemas basicos de restricciones, en principio, sin circularidad en las restricciones, pero tiene problemas al resolver sistemas de restricciones mas complejos, si bien puede ser generalizable.

Dentro de este programa Constraint.Action, se declaran y manejan diferentes tipos de objetos como son: a) La restriccion, que se define como un conjunto (Set) formado por una condicion y una accion (Code condición, Code acción). Su funcionamiento es: si la condicion no se cumple entonces se aplica la accion asociada la restriccion. Por ejemplo, una restricción puede decir: comprueba que en la base de hechos el valor del hecho A es igual a la suma de los valores de los hechos B y C, si esto no es cierto entonces haz que A valga la suma de los valores de B y C. b) El hecho, que es una estructura (Text nombre, Real valor) que se representa mediante un tipo estructurado (Struct) de conjunto. En el punto anterior A, B y C eran hechos. c) La base de hechos, que es un conjunto de hechos.

En Constraint.Action la memoria del metodo recursivo y birecursivo es la propia pila de llamadas y en el metodo iterativo se utiliza como memoria de trabajo un fichero del tipo Bst. En el metodo iterativo, en este fichero Bst se almacena el estado de los valores de los hechos y que van actualizando en cada ciclo de la iteracion donde se produce un cambio en un hecho y este fichero no se actualiza cuando no hay cambios. Por tanto, este fichero Bst representa un estado de la base de hechos donde a cada hecho H se le asigna un valor V. Una posible variante de este programa de resolucion de sistemas de restricciones podria guardar, cada vez, en un fichero Bst de nombre diferente y secuencial. Con ello se podria trazar el proceso completo de solucion. Este programa visualiza por pantalla, para cada caso de test: a) el estado inicial de la base de hachos b) el estado final tras el proceso de aplicacion de las restricciones y c) el metodo iterativo, recursivo o birecursivo que ha empleado. Los 3 casos de prueba, test, construyen una secuencia de Fibonacci (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...) de diferentes longitudes. Este programa Constraint.Action funciona con las versiones de Tol 1.1.5, 1.1.6 y 2.0.1 pero no con la version 1.1.1 al necesitarse la funcion BSTFile(), en la version 1.1.1 se empleaba una funcion, algo similar, llamada Table().

Con el metodo iterativo de Constraint.Action, el paso básico de comprobacion de la condicion asociada a una restriccion y aplicacion, en su caso, de la accion es el siguiente: a) Sea count el contador de restricciones que indica la restriccion que hay que comprobar. b) Sea initBase el estado actual de la base de hechos (memoria de trabajo) leida del fichero en formato Bst. c) Si la restriccion que indica count no se cumple, entonces hay que aplicar la accion de transformacion asociada a dicha restricción y volver a comprobar el conjunto de restricciones desde el principio (esto se hace volviendo a inicializar a 1 la variable de control del numero de restricciones, esto es, count:=1). Notese que el lenguaje Tol es un lenguaje de tipo declarativo donde la declaración de variables se realiza mediante el operador =, esto es que actuan como constantes, solo en algunos casos se utiliza el operador de resaignacion del valor :=, por ejemplo, esto suele suceder en las variables que controlan ciclos de iteracion While(), pero no suele ser necesario en los ciclos For() y EvalSet(). d) Si la restricción que indica la variable count se cumple, entonces no hay que cambiar la base de hechos y se debera proceder a comprobar la siguiente restriccion (esto se hace incrementando la variable de control de las restricciones comprobadas hasta ese momento, count:=count+1). e) Cuando hay algun cambio en la base de hechos se guarda en el fichero Bst el contenido de la base de hechos para que el cambio quede reflejado en la siguiente iteracion.

El metodo recursivo directo de Constraint.Action se basa en un contador de la restriccion actual y los 3 siguientes casos: a) Si no hay mas restricciones que comprobar esta resuelto. b) Si la restriccion actual no se cumple, entonces aplicar la accion y entrar en recursion desde la primera restriccion, el contador a 1. c) Si la restriccion actual se cumple, entonces entrar en recursion con la siguiente restriccion, el contador incrementado en 1, por lo que la combinacion de (b) y (c) producen una recursion muy profunda.

El metodo recursivo basado en 2 funciones de Constraint.Action se basa en un contador de la restriccion actual y los siguientes pasos: a) Busca a partir de la actual la primera restriccion que no se cumpla. b) Si el contador de restricciones encontrado supera al numero de restrucciones entonces es que se cumplen todas y se ha resuelto. c) Si hay una restriccion que no se cumple, entonces aplicar su accion y entrar en recursion desde la primera restriccion, el contador a 1. Con lo que en (a) la recursion no puede ser mayor que el numero de restricciones y se termina con cada busqueda y (c) producen una recursion que aunque profunda solo lo es de cambios, no de pruebas, por lo que es menor que la recursion del metodo recursivo directo.

Árbol de ficheros

Constraint.Action Resuelve sistemas de restricciones por 3 metodos diferentes

- ← make.tol hechos, restricciones y soluciones iterativa y recursivas
- ← make.bat mandato de ejecucion del programa de resolucion de restricciones
- → auxtmp.bst fichero Bst de memoria de base de hechos en el metodo iterativo
- → startlog.txt fichero de log de 3 sistemas de restricciones y 3 metodos
- → constraint_action.pdf documento de funciones de hechos, bases y restricciones

Declaraciones

Constraints

- `Set Constraint(Code condition, Code action)`
  Retorna una nueva restriccion como un par de tipo codigo. Una restricion es un par (condicion, accion) si la condicion no se cumple entonces se aplica la accion.
- `Code ConstraintCondition(Set c)`
  Retorna la condicion (condition) de una restriccion c.
- `Code ConstraintAction(Set c)`
  Retorna la accion (action) de una restriccion c.

Facts

- `Set FactNull`
  Hecho nulo.
- `Text FactName(Set f)`
  Retorna el nombre (name) de un hecho f.
- `Real FactValue(Set f)`
  Retorna el valor (value) de un hecho f.
- `Real FactIsName(Set f, Text name)`
  Retorna TRUE si el nombre del hecho f es name.
- `Real FactIsNull(Set f)`
  Retorna TRUE si se trata del hecho vacio ('',0).
- `Real FactPrint(Set f)`
  Visualiza un hecho y retorna TRUE.

Facts base

- `Text BaseFile`
  Memoria en fichero de la base de hechos.
- `Set BaseFind(Set base, Text name)`
  Retorna el hecho de la base cuyo nombre es name. En la base de hechos solo puede haber un hecho de nombre name. Si hubiera varios (la base mal construida), retorna el primero. Si no hay ningun hecho con dicho nombre retorna el hecho vacio.
- `Set BaseContext(Set base, Text name)`
  Retorna el contexto de un hecho de la base. Se denomina contexto al resto de hechos de la base diferentes del que tiene por nombre name.
- `Set Put(Set base, Text name, Real value)`
  Retorna una base resultado de sobre escribir el hecho de nombre name con el valor values. El resto de hecho seguiran sin cambiar. Si no hubiera ningun hecho de nombre name, lo agrega a la base. Tiene un nombre abreviado para facilitar la escritura de restricciones.
- `Real Get(Set base, Text name)`
  Retorna el valor de un hecho de nombre name de la base. Si el hecho no existe retorna cero. Tiene un nombre abreviado para facilitar la escritura de restricciones.
- `Real BaseTestValue(Set base, Text name, Real value)`
  Retorna TRUE si en base hay un hecho de nombre name con el valor value.

- `Real BaseTest(Set base, Set constraint)`
  Retorna TRUE si en la base de hechos base se cumple la restriccion constraint. Se proporciona una funcion interna Get() que permite escribir las restricciones de forma mas abreviada.
- `Set BaseApplyAction(Set base, Set constraint)`
  Retorna una nueva base resultado de aplicar la parte de accion de la restriccion constraint. No realiza el test de la condicion, aplica directamente la accion.
- `Real BasePrint(Set base)`
  Visualiza todos los hechos una base de hechos y retorna el numero de hechos contenidos en ella.
- `Set BaseWrite(Set base, Text fileName)`
  Escribe en el fichero de nombre fileName una base de hechos.
- `Set BaseRead(Text fileName)`
  Lee del fichero de nombre fileName una base de hechos.
- `Set BaseSolveIterative(Set base, Set constraints)`
  Ciclo iterativo del motor de aplicacion de restricciones y de resolucion. Retorna la base de hechos resuelta.
- `Set BaseSolveRecursive(Set base, Set constraints, Real count)`
  Recursion de aplicacion de restricciones y de resolucion. Retorna la base de hechos resuelta.
- `Real BaseSearch(Set base, Set constraints, Real count)`
  Recursion para buscar el primer incumplimiento desde count incluido. Retorna el numero del primer incumplimiento encontrado.
- `Set BaseSolveBirecursive(Set base, Set constraints, Real count)`
  Recursion de aplicacion de aplicaciones de cambios con una recursion interna de busqueda de restricciones que no se cumplen. Retorna la base de hechos resuelta.
- `Set BaseSolvePrint(Set base, Set constraints, Real method)`
  Llama al ciclo del motor de aplicacion de restricciones imprimiendo, antes y despues, el contenido de la base de hechos. Retorna la base de hechos resuelta.

Pruebas

- `Text datFmt`
  Cambiar el formato de reales para que en la memoria salgan enteros.
- `Set test01`
  Resolver y ver el test 01 recursivamente.
- `Set test02`
  Resolver y ver el test 02 birecursivamente.
- `Set test03`
  Resolver y ver el test 03 iterativamente.

Set Constraint()

//

```
Set Constraint(Code condition,
               Code action)
////////////////////////////////////////////////////////////////////////
{ SetOfCode(condition, action) };
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna una nueva restriccion como un par de tipo codigo.
Una restricion es un par (condicion, accion) si la condicion no se cumple
entonces se aplica la accion.",
Constraint);
////////////////////////////////////////////////////////////////////////
```

Code ConstraintCondition()

```
////////////////////////////////////////////////////////////////////////
Code ConstraintCondition(Set c)
////////////////////////////////////////////////////////////////////////
{ c[1] };
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna la condicion (condition) de una restriccion c.",
ConstraintCondition);
////////////////////////////////////////////////////////////////////////
```

Code ConstraintAction()

```
////////////////////////////////////////////////////////////////////////
Code ConstraintAction(Set c)
////////////////////////////////////////////////////////////////////////
{ c[2] };
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna la accion (action) de una restriccion c.",
ConstraintAction);
////////////////////////////////////////////////////////////////////////
```

Set FactNull

```
////////////////////////////////////////////////////////////////////////
Set FactNull = Fact("",0);
////////////////////////////////////////////////////////////////////////
PutDescription("Hecho nulo.",FactNull);
////////////////////////////////////////////////////////////////////////
```

Text FactName()

```
////////////////////////////////////////////////////////////////////////
Text FactName(Set f)
////////////////////////////////////////////////////////////////////////
{ f->Name };
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna el nombre (name) de un hecho f.",
FactName);
////////////////////////////////////////////////////////////////////////
```

Real FactValue()

```
////////////////////////////////////////////////////////////////////////
```

```
Real FactValue(Set f)
////////////////////////////////////////////////////////////////////////
{ f->Value };
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna el valor (value) de un hecho f.",
FactName);
////////////////////////////////////////////////////////////////////////
```

Real FactIsName()

```
////////////////////////////////////////////////////////////////////////
Real FactIsName(Set  f,
                Text name)
////////////////////////////////////////////////////////////////////////
{ FactName(f)==name };
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna TRUE si el nombre del hecho f es name.",
FactIsName);
////////////////////////////////////////////////////////////////////////
```

Real FactIsNull()

```
////////////////////////////////////////////////////////////////////////
Real FactIsNull(Set f)
////////////////////////////////////////////////////////////////////////
{ And(FactName(f)=="", FactValue(f)==0) };
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna TRUE si se trata del hecho vacio ('',0).",
FactIsNull);
////////////////////////////////////////////////////////////////////////
```

Real FactPrint()

```
////////////////////////////////////////////////////////////////////////
Real FactPrint(Set f)
////////////////////////////////////////////////////////////////////////
{
  Text WriteLn(FactName(f)+": "+FormatReal(FactValue(f), "%.0lf"));
  TRUE
};
////////////////////////////////////////////////////////////////////////
PutDescription(
"Visualiza un hecho y retorna TRUE.",
FactPrint);
////////////////////////////////////////////////////////////////////////
```

Text BaseFile

```
////////////////////////////////////////////////////////////////////////
Text BaseFile = "auxtmp.bst";
////////////////////////////////////////////////////////////////////////
PutDescription("Memoria en fichero de la base de hechos.",BaseFile);
////////////////////////////////////////////////////////////////////////
```

Set BaseFind()

```
////////////////////////////////////////////////////////////////////////////
Set BaseFind(Set  base,
             Text name)
////////////////////////////////////////////////////////////////////////////
{
  Set  s = Select(base, Real (Set f) { FactIsName(f,name) });
  If(Card(s)>0, s[1], FactNull)
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna el hecho de la base cuyo nombre es name.
En la base de hechos solo puede haber un hecho de nombre name.
Si hubiera varios (la base mal construida), retorna el primero.
Si no hay ningun hecho con dicho nombre retorna el hecho vacio.",
BaseFind);
////////////////////////////////////////////////////////////////////////////
```

Set BaseContext()

```
////////////////////////////////////////////////////////////////////////////
Set BaseContext(Set  base,
                Text name)
////////////////////////////////////////////////////////////////////////////
{ Select(base, Real (Set f) { ! FactIsName(f,name) }) };
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna el contexto de un hecho de la base.
Se denomina contexto al resto de hechos de la base diferentes del que tiene
por nombre name.",
BaseFind);
////////////////////////////////////////////////////////////////////////////
```

Set Put()

```
////////////////////////////////////////////////////////////////////////////
Set Put(Set  base,
        Text name,
        Real value)
////////////////////////////////////////////////////////////////////////////
{ SetOfSet(Fact(name,value)) << BaseContext(base,name) };
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna una base resultado de sobre escribir el hecho de nombre name con
el valor values.
El resto de hecho seguiran sin cambiar.
Si no hubiera ningun hecho de nombre name, lo agrega a la base.
Tiene un nombre abreviado para facilitar la escritura de restricciones.",
Put);
////////////////////////////////////////////////////////////////////////////
```

Real Get()

```
////////////////////////////////////////////////////////////////////////////
Real Get(Set  base,
         Text name)
////////////////////////////////////////////////////////////////////////////
{
   Set f = BaseFind(base,name);
   If(FactIsNull(f), 0, FactValue(f))
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna el valor de un hecho de nombre name de la base.
Si el hecho no existe retorna cero.
Tiene un nombre abreviado para facilitar la escritura de restricciones.",
```

```
Get);
////////////////////////////////////////////////////////////////////////
```

Real BaseTestValue()

```
////////////////////////////////////////////////////////////////////////
Real BaseTestValue(Set base,
                   Text name,
                   Real value)
////////////////////////////////////////////////////////////////////////
{
   Set f = BaseFind(base,name);
   If(FactIsNull(f), FALSE, FactValue(f)==value)
};
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna TRUE si en base hay un hecho de nombre name con el valor value.",
BaseTestValue);
////////////////////////////////////////////////////////////////////////
```

Real BaseTest()

```
////////////////////////////////////////////////////////////////////////
Real BaseTest(Set base,
              Set constraint)
////////////////////////////////////////////////////////////////////////
{
   Code condition = ConstraintCondition(constraint);
   condition(base)
};
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna TRUE si en la base de hechos base se cumple la restriccion
constraint.
Se proporciona una funcion interna Get() que permite escribir las
restricciones de forma mas abreviada.",
BaseTest);
////////////////////////////////////////////////////////////////////////
```

Set BaseApplyAction()

```
////////////////////////////////////////////////////////////////////////
Set BaseApplyAction(Set base,
                    Set constraint)
////////////////////////////////////////////////////////////////////////
{
   Code action = ConstraintAction(constraint);
   action(base)
};
////////////////////////////////////////////////////////////////////////
PutDescription(
"Retorna una nueva base resultado de aplicar la parte de accion de la
restriccion constraint.
No realiza el test de la condicion, aplica directamente la accion.",
BaseApplyAction);
////////////////////////////////////////////////////////////////////////
```

Real BasePrint()

```
////////////////////////////////////////////////////////////////////////
Real BasePrint(Set base)
////////////////////////////////////////////////////////////////////////
```

```
{ Card(EvalSet(base, FactPrint)) };
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Visualiza todos los hechos una base de hechos y retorna el numero de hechos
contenidos en ella.",
BasePrint);
////////////////////////////////////////////////////////////////////////////
```

Set BaseWrite()

```
////////////////////////////////////////////////////////////////////////////
Set BaseWrite(Set  base,
              Text fileName)
////////////////////////////////////////////////////////////////////////////
{ BSTFile(base, fileName, "Fact;Name;Value") };
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Escribe en el fichero de nombre fileName una base de hechos.",
BaseWrite);
////////////////////////////////////////////////////////////////////////////
```

Set BaseRead()

```
////////////////////////////////////////////////////////////////////////////
Set BaseRead(Text fileName)
////////////////////////////////////////////////////////////////////////////
{ IncludeBST(fileName) };
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Lee del fichero de nombre fileName una base de hechos.",
BaseRead);
////////////////////////////////////////////////////////////////////////////
```

Set BaseSolveIterative()

```
////////////////////////////////////////////////////////////////////////////
Set BaseSolveIterative(Set base,
                       Set constraints)
////////////////////////////////////////////////////////////////////////////
{
  Real count = 1;
  Real maxCount = Card(constraints);

  Set BaseWrite(base,BaseFile);

  Real While(count <= maxCount,
  {
    Set initBase = BaseRead(BaseFile);
    If(! BaseTest(initBase, constraints[count]),
    {
      Set changedBase = BaseApplyAction(initBase, constraints[count]);
      Set BaseWrite(changedBase, BaseFile); // Escribir ahora que hay cambios
      Real(count := 1);
      TRUE  // Con cambios
    },
    {
      Real(count := count + 1);
      FALSE // Sin cambios
    })
  });

  BaseRead(BaseFile) // Retorna la ultima solucion del fichero
};
////////////////////////////////////////////////////////////////////////////
```

```
PutDescription(
"Ciclo iterativo del motor de aplicacion de restricciones y de resolucion.
Retorna la base de hechos resuelta.",
BaseSolveIterative);
////////////////////////////////////////////////////////////////////////////
```

Set BaseSolveRecursive()

```
////////////////////////////////////////////////////////////////////////////
Set BaseSolveRecursive(Set  base,
                       Set  constraints,
                       Real count)
////////////////////////////////////////////////////////////////////////////
{
  If(count > Card(constraints), base,             // Resuelto
     If(! BaseTest(base, constraints[count]),     // Si no se cumple -> accion
          BaseSolveRecursive(BaseApplyAction(base,constraints[count]),
                             constraints, 1),
          BaseSolveRecursive(base, constraints, count+1))) // Se va cumpliendo
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Recursion de aplicacion de restricciones y de resolucion.
Retorna la base de hechos resuelta.",
BaseSolveRecursive);
////////////////////////////////////////////////////////////////////////////
```

Real BaseSearch()

```
////////////////////////////////////////////////////////////////////////////
Real BaseSearch(Set  base,
                Set  constraints,
                Real count)
////////////////////////////////////////////////////////////////////////////
{
  Case(
    count > Card(constraints),                 count,    // Esta perfecto
    !BaseTest(base, constraints[count]),       count,    // Esta no se cumple
    TRUE,       BaseSearch(base, constraints, count+1)) // Buscar el siguiente
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Recursion para buscar el primer incumplimiento desde count incluido.
Retorna el numero del primer incumplimiento encontrado.",
BaseSearch);
////////////////////////////////////////////////////////////////////////////
```

Set BaseSolveBirecursive()

```
////////////////////////////////////////////////////////////////////////////
Set BaseSolveBirecursive(Set  base,
                         Set  constraints,
                         Real count)
////////////////////////////////////////////////////////////////////////////
{
  Real newCount = BaseSearch(base,constraints,count); // Busca incumplimiento

  If(newCount > Card(constraints), base,   // Resuelto
     BaseSolveBirecursive(BaseApplyAction(base, constraints[newCount]),
                          constraints, 1)) // Tras la accion volver a empezar
};
////////////////////////////////////////////////////////////////////////////
PutDescription(
"Recursion de aplicacion de aplicaciones de cambios con una recursion interna
de busqueda de restricciones que no se cumplen.
Retorna la base de hechos resuelta.",
```

```
BaseSolveBirecursive);
////////////////////////////////////////////////////////////////////////
```

Set BaseSolvePrint()

```
////////////////////////////////////////////////////////////////////////
Set BaseSolvePrint(Set  base,        // Base de hechos
                   Set  constraints, // Restricciones
                   Real method)      // 1 iterativo, 2 recursivo, 3 birecurs
////////////////////////////////////////////////////////////////////////
{
  Text WriteLn(Repeat("_", 78)+"\nInitial fact base:");
  Real ini = BasePrint(base); // Ver el punto de partida

  Set  sol = Case(
    method == 1, BaseSolveIterative  (base, constraints),     // Itera
    method == 2, BaseSolveRecursive  (base, constraints, 1),  // Recursivo
    TRUE,        BaseSolveBirecursive(base, constraints, 1)); // Birecursivo

  Text mode = Case(
    method == 1, "Iterarive",
    method == 2, "Recursive",
    TRUE,        "Birecursive");

  Text WriteLn(Repeat("_", 78)+"\nSolved: " + mode);
  Real end = BasePrint(sol);  // Ver los resultados

  sol
};
////////////////////////////////////////////////////////////////////////
PutDescription(
"Llama al ciclo del motor de aplicacion de restricciones imprimiendo,
antes y despues, el contenido de la base de hechos.
Retorna la base de hechos resuelta.",
BaseSolvePrint);
////////////////////////////////////////////////////////////////////////
```

Text datFmt

```
////////////////////////////////////////////////////////////////////////
Text datFmt = PutRealFormat("%4.0lf");
////////////////////////////////////////////////////////////////////////
PutDescription(
"Cambiar el formato de reales para que en la memoria salgan enteros.",
datFmt);
////////////////////////////////////////////////////////////////////////
```

Set test01

```
////////////////////////////////////////////////////////////////////////
Set  test01 = If(FALSE, Empty, // Cambiar TRUE/FALSE para ejecutar
{
  Text WriteLn("\nConstraint.Action make: test 01");

  Set base01 =  // Base de 9 hechos del test 01
  [[ Fact("A", 1),
     Fact("B", 1),
     Fact("C", 1),
     Fact("D", 1),
     Fact("E", 1),
     Fact("F", 1),
     Fact("G", 1),
     Fact("H", 1),
     Fact("I", 1)
```

```
  ]];

  Set cons01 = // Resctricciones del test 01
  [[
    Constraint
      (Real (Set b) { Get(b,"A") == Real(Get(b,"B")+Get(b,"C")) },
       Set  (Set b) { Put(b,"A",    Get(b,"B")+Get(b,"C")) }),
    Constraint
      (Real (Set b) { Get(b,"B") == Real(Get(b,"C")+Get(b,"D")) },
       Set  (Set b) { Put(b,"B",    Get(b,"C")+Get(b,"D")) }),
    Constraint
      (Real (Set b) { Get(b,"C") == Real(Get(b,"D")+Get(b,"E")) },
       Set  (Set b) { Put(b,"C",    Get(b,"D")+Get(b,"E")) }),
    Constraint
      (Real (Set b) { Get(b,"D") == Real(Get(b,"E")+Get(b,"F")) },
       Set  (Set b) { Put(b,"D",    Get(b,"E")+Get(b,"F")) }),
    Constraint
      (Real (Set b) { Get(b,"E") == Real(Get(b,"F")+Get(b,"G")) },
       Set  (Set b) { Put(b,"E",    Get(b,"F")+Get(b,"G")) }),
    Constraint
      (Real (Set b) { Get(b,"F") == Real(Get(b,"G")+Get(b,"H")) },
       Set  (Set b) { Put(b,"F",    Get(b,"G")+Get(b,"H")) }),
    Constraint
      (Real (Set b) { Get(b,"G") == Real(Get(b,"H")+Get(b,"I")) },
       Set  (Set b) { Put(b,"G",    Get(b,"H")+Get(b,"I")) })
  ]];

  BaseSolvePrint(base01, cons01, 2) // Recursive
});
////////////////////////////////////////////////////////////////////////////
PutDescription("Resolver y ver el test 01 recursivamente.", test01);
////////////////////////////////////////////////////////////////////////////
```

Set test02

```
////////////////////////////////////////////////////////////////////////////
Set  test02 = If(FALSE, Empty, // Cambiar TRUE/FALSE para ejecutar
{
  Text WriteLn("\nConstraint.Action make: test 02");

  Set base02 = // Base de 11 hechos del test 02
  [[ Fact("A", 1),
     Fact("B", 1),
     Fact("C", 1),
     Fact("D", 1),
     Fact("E", 1),
     Fact("F", 1),
     Fact("G", 1),
     Fact("H", 1),
     Fact("I", 1),
     Fact("J", 1),
     Fact("K", 1)
  ]];

  Set cons02 = // Resctricciones del test 02
  [[
    Constraint
      (Real (Set b) { Get(b,"A") == Real(Get(b,"B")+Get(b,"C")) },
       Set  (Set b) { Put(b,"A",    Get(b,"B")+Get(b,"C")) }),
    Constraint
      (Real (Set b) { Get(b,"B") == Real(Get(b,"C")+Get(b,"D")) },
       Set  (Set b) { Put(b,"B",    Get(b,"C")+Get(b,"D")) }),
    Constraint
      (Real (Set b) { Get(b,"C") == Real(Get(b,"D")+Get(b,"E")) },
       Set  (Set b) { Put(b,"C",    Get(b,"D")+Get(b,"E")) }),
    Constraint
      (Real (Set b) { Get(b,"D") == Real(Get(b,"E")+Get(b,"F")) },
       Set  (Set b) { Put(b,"D",    Get(b,"E")+Get(b,"F")) }),
    Constraint
      (Real (Set b) { Get(b,"E") == Real(Get(b,"F")+Get(b,"G")) },
       Set  (Set b) { Put(b,"E",    Get(b,"F")+Get(b,"G")) }),
    Constraint
```

```
    (Real (Set b) { Get(b,"F") == Real(Get(b,"G")+Get(b,"H")) },
     Set  (Set b) { Put(b,"F",    Get(b,"G")+Get(b,"H")) }),
  Constraint
    (Real (Set b) { Get(b,"G") == Real(Get(b,"H")+Get(b,"I")) },
     Set  (Set b) { Put(b,"G",    Get(b,"H")+Get(b,"I")) }),
  Constraint
    (Real (Set b) { Get(b,"G") == Real(Get(b,"H")+Get(b,"I")) },
     Set  (Set b) { Put(b,"G",    Get(b,"H")+Get(b,"I")) }),
  Constraint
    (Real (Set b) { Get(b,"H") == Real(Get(b,"I")+Get(b,"J")) },
     Set  (Set b) { Put(b,"H",    Get(b,"I")+Get(b,"J")) }),
  Constraint
    (Real (Set b) { Get(b,"I") == Real(Get(b,"J")+Get(b,"K")) },
     Set  (Set b) { Put(b,"I",    Get(b,"J")+Get(b,"K")) })
]];

BaseSolvePrint(base02, cons02, 3) // Birecursive
});
////////////////////////////////////////////////////////////////////////////
PutDescription("Resolver y ver el test 02 birecursivamente.", test02);
////////////////////////////////////////////////////////////////////////////
```

Set test03

```
////////////////////////////////////////////////////////////////////////////
Set  test03 = If(FALSE, Empty, // Cambiar TRUE/FALSE para ejecutar
{
  Text WriteLn("\nConstraint.Action make: test 03");

  Set base03 = // Base de 15 hechos del test 03
  [[ Fact("A", 1),
     Fact("B", 1),
     Fact("C", 1),
     Fact("D", 1),
     Fact("E", 1),
     Fact("F", 1),
     Fact("G", 1),
     Fact("H", 1),
     Fact("I", 1),
     Fact("J", 1),
     Fact("K", 1),
     Fact("L", 1),
     Fact("M", 1),
     Fact("N", 1),
     Fact("O", 1)
  ]];

  Set cons03 = // Resctricciones del test 03
  [[
    Constraint
      (Real (Set b) { Get(b,"A") == Real(Get(b,"B")+Get(b,"C")) },
       Set  (Set b) { Put(b,"A",    Get(b,"B")+Get(b,"C")) }),
    Constraint
      (Real (Set b) { Get(b,"B") == Real(Get(b,"C")+Get(b,"D")) },
       Set  (Set b) { Put(b,"B",    Get(b,"C")+Get(b,"D")) }),
    Constraint
      (Real (Set b) { Get(b,"C") == Real(Get(b,"D")+Get(b,"E")) },
       Set  (Set b) { Put(b,"C",    Get(b,"D")+Get(b,"E")) }),
    Constraint
      (Real (Set b) { Get(b,"D") == Real(Get(b,"E")+Get(b,"F")) },
       Set  (Set b) { Put(b,"D",    Get(b,"E")+Get(b,"F")) }),
    Constraint
      (Real (Set b) { Get(b,"E") == Real(Get(b,"F")+Get(b,"G")) },
       Set  (Set b) { Put(b,"E",    Get(b,"F")+Get(b,"G")) }),
    Constraint
      (Real (Set b) { Get(b,"F") == Real(Get(b,"G")+Get(b,"H")) },
       Set  (Set b) { Put(b,"F",    Get(b,"G")+Get(b,"H")) }),
    Constraint
      (Real (Set b) { Get(b,"G") == Real(Get(b,"H")+Get(b,"I")) },
       Set  (Set b) { Put(b,"G",    Get(b,"H")+Get(b,"I")) }),
    Constraint
      (Real (Set b) { Get(b,"G") == Real(Get(b,"H")+Get(b,"I")) },
```

```
      Set  (Set b) { Put(b,"G",    Get(b,"H")+Get(b,"I")) }),
    Constraint
      (Real (Set b) { Get(b,"H") == Real(Get(b,"I")+Get(b,"J")) },
       Set  (Set b) { Put(b,"H",    Get(b,"I")+Get(b,"J")) }),
    Constraint
      (Real (Set b) { Get(b,"I") == Real(Get(b,"J")+Get(b,"K")) },
       Set  (Set b) { Put(b,"I",    Get(b,"J")+Get(b,"K")) }),
    Constraint
      (Real (Set b) { Get(b,"J") == Real(Get(b,"K")+Get(b,"L")) },
       Set  (Set b) { Put(b,"J",    Get(b,"K")+Get(b,"L")) }),
    Constraint
      (Real (Set b) { Get(b,"K") == Real(Get(b,"L")+Get(b,"M")) },
       Set  (Set b) { Put(b,"K",    Get(b,"L")+Get(b,"M")) }),
    Constraint
      (Real (Set b) { Get(b,"L") == Real(Get(b,"M")+Get(b,"N")) },
       Set  (Set b) { Put(b,"L",    Get(b,"M")+Get(b,"N")) }),
    Constraint
      (Real (Set b) { Get(b,"M") == Real(Get(b,"N")+Get(b,"O")) },
       Set  (Set b) { Put(b,"M",    Get(b,"N")+Get(b,"O")) })
  ]];

  BaseSolvePrint(base03, cons03, 1) // Iterative
});
////////////////////////////////////////////////////////////////////////////
PutDescription("Resolver y ver el test 03 iterativamente.", test03);
////////////////////////////////////////////////////////////////////////////
```