Universidad Simón Bolívar
Departamento de Computación y T.I.
CI–5437 Inteligencia Artificial I
Abril–Julio 2010

Proyecto II – Planificación y SAT – 10 %

versión 1.0

Entrega: Viernes 11ra, antes de las 12 del medio día.

El objetivo del proyecto es modificar un planificador basado en SAT para que pueda recibir restricciones (*constraints*) sobre los estados definidas como condiciones sobre los atomos. Ahora definiremos parte de sintaxis básica de un archivo PDDL y que parte corresponderá a las restricciones. Más adelante indicaremos como deberá hacer dichos cambios en el planificador.

```
<domain>          ::=  (define (domain <name>)
                         <types-def>
                         <predicates-def>
                         <constraints>
                         <actions-def>*)
<constraints>     ::=  (:constraints <con-GD>)
<con-GD>          ::=  <con-GD-simple>
                       (and <con-GD-simple>+)
<con-GD-simple>   ::=  (at end <GD>)
                       (always <GD>)
                       (sometime <GD>)
                       (at-most-once <GD>)
                       (sometimes-after <GD> <GD>)
                       (sometimes-before <GD> <GD>)
<GD>               :=  <literal>
                       (and <GD>*)
                       (or <GD>*)
                       (imply <GD> <GD>)
<literal>         ::=  <atomic-formula>
                       (not <atomic-formula>)
<atomic-formula>  ::=  (predicate <term>*)
<term>            ::=  <name>
                       <variable>
<variable>        ::=  ?<name>

<problem>         ::=  (define (problem <name>)
                         (:domain <name>)
                         <object-declaration>
                         <init>
                         <goal>
                         <constraints>)
```

Se ha restringido la definición de constraints del PDDL3.0 a un lenguaje que sea facil de implementar en el planificador basado en sat. Nótese que arriba quedan algunos no terminales por definir como `<types-def>`, `<predicates-def>`, etc, que ya están implementados correctamente en

el planificador. Para ver el significado de `(always...)`, `(sometime...)`, etc, y para más detalles ver `http://zeus.ing.unibs.it/ipc-5/bnf.pdf`.

Recordemos que para un problema de planificación P y para un número de pasos a ejecutar n, un planificador basado en SAT genera variables proposicionales p_i para cada $0 \leq i \leq n$. Así, la restricción `(always p)` puede ser traducida como agregar $n+1$ clausulas de un sólo literal p_i. En cambio la restricción `(sometime p)` puede ser codificada usando una única clausula $p_0 \vee p_1 \vee \ldots \vee p_n$.

A fines de simplificar la entrega, puede suponer que $<GD>$ sólo puede ser un literal. Queda como **extra-crédito** los otros casos con operadores lógicos.

> *En ese escenario, una posible dificultad es que* `<GD>` *puede ser más complejo que un simple literal. En ese caso recuerden que para una formula compleja tipo* (and p q)*, siempre puede crearse una variable auxiliar $z \equiv p \wedge q$, que traducido a CNF quedaría: $(\neg z \vee p) \wedge (\neg z \vee p) \wedge (\neg p \vee \neg q \vee z)$. Luego puede usarse z en las construcciones* `always`*,* `sometime`*, etc.*

Actividades:

1. Modificar el planificador `num2sat`, basado en SAT, para que involucre los cambios anteriores. Dicho planificador utiliza una codificación diferente a la que vimos en clase: por ejemplo no usa no-ops.
   - El planificador puede encontrarse en `http://www.ldc.usb.ve/~hlp/share/satplan-cond-effects-2009.zip`.
   - El ejecutable que crea se llama `num2sat`.
   - Propongo dos posibles estrategias para implementar la tarea, que pueden ser incluso combinadas. Los cambios son relativamente sencillos y no es necesario entender todo el código en ningún caso.
     - Modificar el parser, y la generación de clausulas (en el archivo CNF.c).
     - Procesar los pddls antes de pasarlos al planner, y luego en el planner llamar a otro programa dentro que genera las clausulas.
       - Para esta opción, si quieren pueden mirar memory-less.py, y que dentro del planner se llama a minisat+mutex.py. Sin importar lo que hacen concretamente, en memory-less.py se parsea muy informalmente los PDDLs y se generan datos que son usados luego por minisat+mutex.py.
       - No se responderá más de UNA pregunta por grupo sobre esta opción.

Requerimientos técnicos:

1. No debe reducir o simplificar la sintaxis de PDDL soportada por num2sat.
2. Si necesita hacer algún otro programa auxiliar, puede hacerlo en el lenguaje que quiera.
3. Su proyecto debe funcionar en las maquinas Linux del LDC, incluyendo la versión del lenguaje de programación que esté allí instalada.
4. Su ejecutable debe llamarse `num2sat` y deberá recibir los mismos parámetros que recibía `num2sat`. Puede hace algún programa que a su vez llame al planificador, pero es importante que para usar el resultado de su tarea el planificador deba usarse un programa llamado *num2sat*.
5. Debe entregar un archivo `.tar.gz` que contenga al menos:

informe.pdf que describa muy resumidamente las decisiones tomadas para hacer su proyecto. Debe comentar sus decisiones sobre estructuras de datos, y una discusión sobre el impacto esperado de esos cambios en el proyecto.

README.txt que indique los requerimientos técnicos de su programa y otra información relevante sobre los archivos del `.tar.gz`

Makefile tal que `make` compile su programa.

¿Cómo comenzar a trabajar rápidamente?

1. Baje y compile el num2sat de `http://www.ldc.usb.ve/~hlp/share/satplan-cond-effects-2009.zip`.

2. Baje algunos PDDLs usados en la International Planning Competition y trate de resolverlo. Se recomienda probar en problemas muy pequeños. El sitio de la IPC4 es `http://www.tzi.de/~edelkamp/ipc-4/`.