Universidad Zaragoza

Anexos Proyecto Fin de Carrera

Creación de un Agente basado en la arquitectura SOAR

Autora

Teresa Albajar Lafarga

Directores

Dr. Francisco José Serón

Dr. Manuel G. Bedía

Departamento de Informática e Ingeniería de Sistemas

Escuela de Ingeniería y Arquitectura

2015

Anexo A

Programación de Reglas

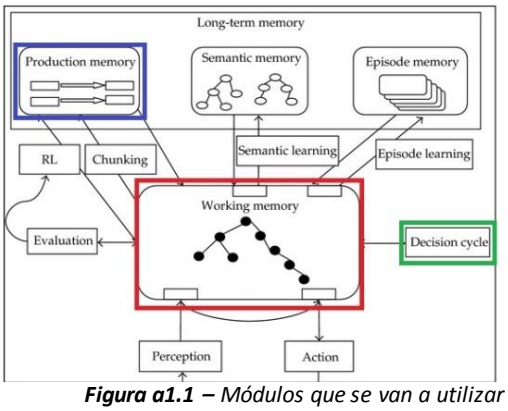


Figura a1.1 *– Módulos que se van a utilizar*

Se va a utilizar un ejemplo para mostrar cómo realizar programación basada en reglas. Para este ejemplo utilizamos 3 sistemas de SOAR (ver Figura a1.1), la Working memory (encuadrada en rojo), la Procedural memory (encuadrada en azul, también llamada production memory), y el módulo de decisión (en verde) explicadas previamente en la introducción a SOAR. Nos basaremos en cómo se simboliza la información en la Working memory (representación simbólica en forma de árbol) para escribir las reglas necesarias que estarán en la Procedural memory. De ellas algunas serán posibles opciones en cada ciclo y el módulo de decisión elegirá cuál debe ejecutar sus acciones.

El sistema cognitivo SOAR se "programa" utilizando reglas. Cada regla se compone de dos partes diferenciadas por el símbolo "→". La primera corresponde a las precondiciones de la regla, lo que se debe cumplir para que se pueda utilizar; también en esta zona solemos tener la asignación de algunas variables que serán utilizadas después. En la segunda parte tenemos el operador a aplicar (o los cambios a realizar en la Working memory debido a la aplicación de la regla en cuestión). Ejemplo:

```
water-jug*propose*initialize-water-jug
Si no hay una tarea seleccionada, proponer el operador initialize-
water-jug.

Sp {water-jug*propose*initialize-water-jug
      (state <s> ^superstate nil)
      -(<s> ^name)
-->
     (<s>  ^operator <o> +)
     (<o> ^name initialize-water-jug) }
```

En la literatura se utiliza el nombre de la regla, en este caso "water-jug*propose*initialize-water-jug", más una pequeña descripción. Esta descripción tiene la misma forma que la regla, "Si… (Precondición) entonces … (operador o cambios)", y sirve para facilitar la interpretación de la misma.

Todas las reglas empiezan con las letras "sp" (soar production), un espacio y abrir llave "{". Toda la regla estará escrita dentro de esa llave que se debe cerrar al final. El nombre de la regla ira inmediatamente después de la primera llave, sin espacios, puede contener letras, números, "*" y "-" que se utilizan para separar las palabras. No importa la longitud del nombre de la regla ya que solo sirve al programador para diferenciarlas; sin embargo deben ser nombres diferentes para cada una. Después se

hace salto de línea y empiezan las precondiciones, lo que entra en el "Si…". Aquí tenemos pequeñas unidades, las cuales van siempre entre paréntesis y tienen una estructura.

Lo primero para entender la estructura de las unidades de condiciones (que también funciona para las acciones en la segunda parte) es entender cómo se estructura la Working memory. En la Working memory tenemos el estado del agente en un momento concreto, esto incluye diversas entidades con atributos que enlazan a un valor o a otra entidad creando un árbol (ver Figura a1.2).

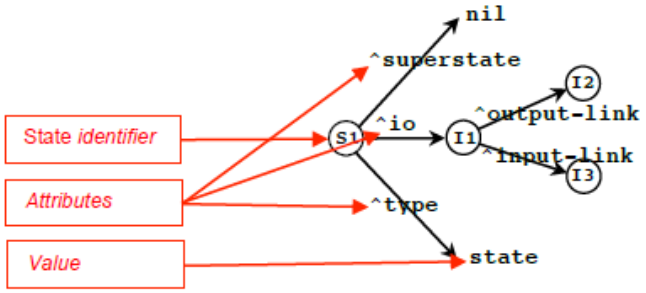


Figura a1.2- *Diagrama del estado inicial*

Este es un ejemplo del estado inicial que tendría cualquier agente. SOAR da un nombre a cada entidad, en este caso S1 es nuestro estado inicial y tiene 3 atributos (3 líneas que salen de él). El super-estado (que es el estado del cual procede, en este caso está a nulo, ya que este es el estado inicial), la entrada-salida de información y el tipo de entidad que es. Los atributos, cuyo símbolo son las líneas se escriben precedidas de un "^" para diferenciarlas de valores u entidades, y son solo una palabra que expresa qué es el valor que está al final de la línea. Los utilizaremos para poder acceder a los valores de cada entidad. En este caso el atributo ^superstate termina en un valor (nil), sin embargo el atributo ^io termina en otra entidad que a su vez tiene su atributo ^output-link e ^input-link.

Visto esto una unidad de condición o acción será de la forma general:

(Identificador ^atributo identificador)

Sin embargo la primera unidad de condición siempre comienza con la palabra "state" como vemos en nuestro ejemplo anterior: *(state <s> ^superstate nil)* ya que para acceder a cualquier entidad se accede por el nodo raíz del árbol, por lo que siempre accederemos por el estado inicial con la palabra state y guardamos en <s> cuál es. Los identificadores que usamos en las unidades se distinguen por tener la forma "<nombre>". Esto quiere decir que se les ha asignado un valor en algún momento y mediante los atributos se puede acceder a las entidades o valores que "cuelguen" de él en el árbol de la Working memory. Por eso aquí <s> será nuestro estado inicial y por lo tanto podemos acceder a su atributo ^superstate. El segundo

identificador de la unidad, tras el atributo actúa de la siguiente forma: si escribimos un valor estamos escribiendo una condición para la regla, si tenemos *(<s> ^color rojo)* estamos forzando a que esta regla solo pueda ejecutarse si se cumple que <s> tiene un atributo que se llama ^color y que es rojo. Si tenemos *(<s> ^color <c>)* no se comporta como una condición sino que guarda en <c> el valor del atributo ^color que cuelgue de <s> y que nos ayudará posteriormente en la norma para escribir alguna condición. Por ultimo nos encontramos la opción *(<s> ^name)* lo cual pide como condición que <s> tenga un atributo ^name, sin importar su valor sin embargo en nuestro ejemplo ésta condición está precedida por un signo "-" el cual actúa como una negación, lo cual significa que quieres que <s> **no** tenga el atributo ^name, que no haya nada con ese nombre. Por último aunque normalmente se sigue esta estructura se puede utilizar esta: *(<s> ^color <c> ^tipo <t> ^superficie <su>)* todo seguido, que equivale a: *(<s> ^color <c>) (<s> ^tipo <t>) (<s> ^superficie <su>)* en 3 líneas diferentes.

Una vez acabamos con las condiciones y ponemos "→" entramos en la zona "entonces…" de la regla. Aquí modificamos la working memory o proponemos un operador. Por ejemplo:

```
(<s>  ^operator <o> +)
(<o> ^name initialize-water-jug)
```

En la primera línea generamos un atributo (o modificamos) ^operator de valor <o> (que aún no lo tenemos y poniendo esto le decimos a SOAR que sea él quien le dé el nombre y directamente nos lo asigne a ese identificador) y le ponemos un "+". Los operadores deben tener un valor para saber, al ser elegidos, si son aceptable +, better >, worse <, best >>, indifferent =, reject < … (estos valores se explican previamente). Este valor se escribe aquí tras ponerle al estado el operador. A veces se pueden usar 2 símbolos como += de forma que decimos que es aceptable pero a la vez indiferente. Si nos encontramos con varios operadores a elegir que tienen esta forma sabemos que SOAR elegirá al azar entre ellos ya que tienen el rasgo "indifferent". Tras esto al operador <o> cuyo nombre como entidad en la Working memory habrá elegido SOAR le damos su atributo ^name necesario para el programador y para poder seleccionar posteriores reglas. Aquí le ponemos el valor "initialize-water-jug".

A veces, como veremos más adelante, es necesario utilizar ciertas formulas sencillas matemáticas en nuestras reglas. Para ello se programa de forma similar a Lisp[1]. Es interesante saber que la primera implementación de la arquitectura SOAR fue construida usando Lisp. Ejemplo:

(<j> ^empty (- <v> <c>))

[1] Enlace externo: http://es.wikipedia.org/wiki/Lisp

Nuestra fórmula tendrá que estar siempre entre paréntesis. Aquí tenemos que al atributo ^empty de <j> le vamos a asignar la resta entre el valor en <v> y el valor en <c>, previamente asignados.

Otras cosas útiles que podemos escribir en nuestras unidades de condición o acción son, por ejemplo, escribir un mensaje por "pantalla" para el debugger o usar el comando "halt" que para y finaliza el agente (si llega a su meta). Ejemplo:

```
(write |Hello world|)
(halt)
```

Para usar el write debemos escribir el mensaje entre "|". Este comando es útil para la revisión y comprensión del código para el programador, pero raramente se utiliza en agentes finales. Halt puede ser usado en agentes finales para finalizar la actividad, sin embargo, ya que SOAR está pensado para sistemas que no hagan sólo una tarea y acaben, sino para agentes que persistan, sobrevivan y aprendan en el tiempo, no es necesario ni obligatorio que exista una regla de cada agente con un (halt).

Creación y análisis de un agente sencillo

El programa de Visual Soar es un "editor de texto" que nos ayuda a generar los ficheros necesarios para hacer un programa Soar de reglas. Se puede hacer en cualquier otro editor, pero este genera las carpetas y sub-carpetas más eficientes y con la estructura correcta, además de ayudar con cierta estructura del texto.

Se va a utilizar el problema de Water-jug como ejemplo de uso y para entender bien como programar esta tarea, en vista a entender en funcionamiento posterior de los sistemas de aprendizaje. El problema que se va a ver aquí es el más simple; para aplicar el aprendizaje se tendrán que modificar algunas cosas que se explicarán en el anexo dedicado a ello.

Pasos a seguir utilizando el VisualSoar:

Empezamos por hacer click en *File -> New Project* y nos aparecerá una ventana con el nombre de *New Agent*. Aquí se escribirá el nombre del problema y se elige la carpeta donde guardar el proyecto; lo llamaremos "water-jug".

Se ve una ventana a nuestra izquierda que contendrá 5 ítems con estructura de árbol. Esta ventana se llama *operator window*. En la raíz se encuentra el nombre del proyecto y los otros 4 ítems por debajo son los ficheros por defecto que se crean automáticamente. Se hablará sobre ellos más adelante. Ficheros como estos contendrán las reglas que se escriban para el agente; estas reglas se agruparan en ficheros de diferentes tipos dependiendo de su función. En esta ventana se controla el modo en que las reglas se agrupan entre sí. EL sistema actual es totalmente arbitrario

para el sistema pero es mucho más fácil para el programador a la hora de mantener y hacer debugg del código.

Una utilidad importante del programa es poder visualizar la estructura en forma de árbol de la Working memory con lo que vamos escribiendo en el agente. Esto lo podemos ver haciendo click derecho en la raíz (llamada 'water-jug') y en *Open Datamap (ver Figura a1.3)*. No solo permite ver la estructura sino que permite hacer test de funcionamiento para asegurar que el código sigue una estructura correcta, encontrando fallos de escritura o estructuras mal puestas.

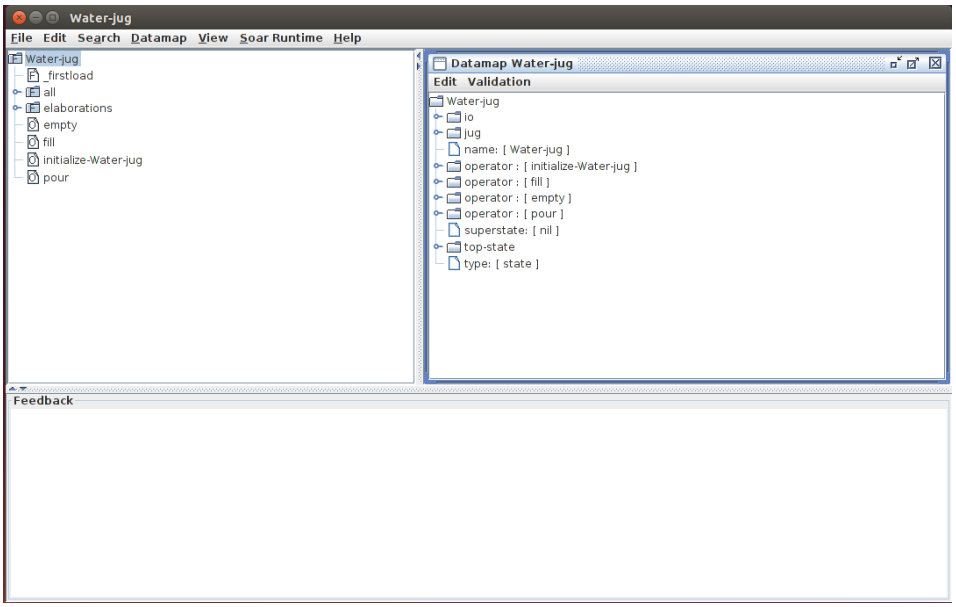


Figura a1.3 – *Ventana del VisualSoar con Datamap abierto*

El problema:

> Se tienen dos cubos vacíos de agua, uno con una capacidad de 5 litros y otro de 3 litros. Hay un pozo con agua ilimitada que se puede usar para llenar completamente los cubos. También se pueden vaciar o pasar el agua de un cubo a otro. No hay marcas en los cubos para niveles intermedios de capacidad. La meta es conseguir tener 1 litro de agua en el cubo de 3 litros.

La primera aproximación al problema debe ser analizar el "espacio del problema" (*problem-space)*, determinado por los objetos manipulables (los dos cubos) y sus valores posibles (de 0 a 5 litros).

Estas van a ser las entidades que se irán modificando en la Working memory, por lo tanto generaremos una entidad cubo, con el atributo capacidad del cubo, cantidad llenada y espacio vacío (para facilitar las reglas).

Hay que definir un estado inicial. En este caso el estado inicial será tener dos entidades cubo con el atributo ^capacidad a 5 y el otro a 3 y el atributo ^cantidad a 0. El estado final también debe ser definido, tendremos que crear una regla que compruebe si

^cantidad de cubo de ^capacidad 3 es igual a 1. Si es así se acaba el programa (se alcanza la solución).

Los operadores principales del problema son los cambios y acciones que podemos ejecutar para intentar llegar al estado final. Para este ejemplo concreto necesitamos uno que sea "llenar" uno de los cubos con el agua del pozo, este operador lo llamaremos "fill". Otro que sea "vaciar" el contenido del cubo de forma que se quede a ^cantidad 0, a este lo llamaremos "empty". Y por último uno que coja el agua de un cubo y la vierta en el otro cubo siempre y cuando cumpla que el cubo a verter tenga agua y que el cubo donde se vierta no esté completamente lleno. Para este caso se hará el cálculo según las cantidades y capacidades de los cubos del estado final tras aplicar esta regla y se modificaran los valores de los atributos en consecuencia. Esta regla la llamaremos "pour".

En SOAR es importante tener en cuenta un conocimiento llamado *search control* que controle que los operadores aplicados no hacen cosas poco óptimas como vaciar un cubo que acabas de llenar en el paso justo anterior. Se crearan unas reglas concretas que regulen este comportamiento en el apartado del mismo nombre.

Vamos a usar una representación sencilla para los estados del problema: *5:0,3:0* representara el cubo de 5 litros con 0 litros de agua y el cubo de 3 litros con 0 litros de agua.

Tenemos el siguiente diagrama de estados que muestra todas las posibles combinaciones y marca los dos estados posibles finales y como se llega a ellos (ver Figura a1.4). La solución óptima del problema es la siguiente:

(5:0,3:0) Llenar el cubo de 3l
(5:0,3:3) Echar el agua del cubo de 3l en el de 5l
(5:3,3:0) Llenar el cubo de 3l
(5:3,3:3) Echar el agua del cubo de 3l en el de 5l
(5:5,3:1) Estado deseado.

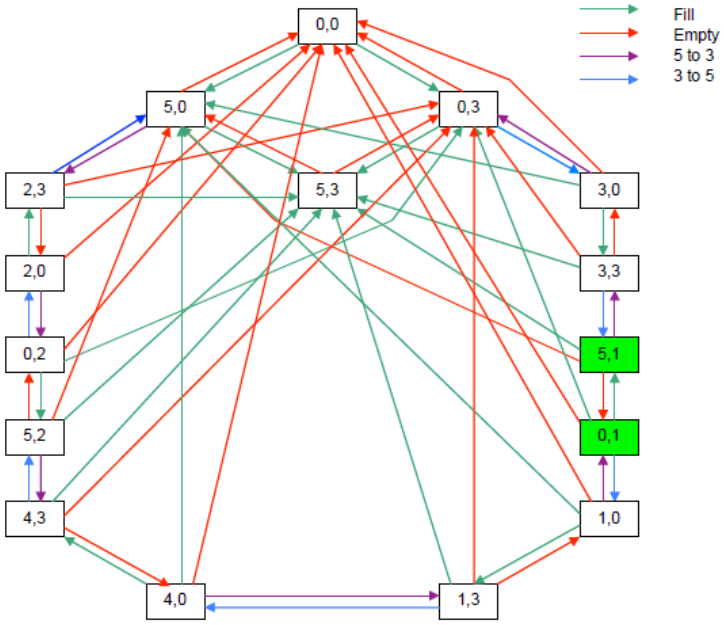


Figura a1.4

Las flechas morada y azul representan los operadores "pour" en la versión de echar el contenido de 5 en 3 y echar el contenido de 3 en 5 respectivamente. No se diferencia el llenado del cubo de 5l o de 3l ni el vaciado con colores.

Programación de las Reglas

```
water-jug*propose*initialize-water-jug
Si no hay una tarea seleccionada, propón el operador initialize-water-jug.

Sp {water-jug*propose*initialize-water-jug
      (state <s> ^superstate nil)
      -(<s> ^name)
-->
     (<s>  ^operator <o> +)
     (<o> ^name initialize-water-jug) }
```

En VisualSoar el último objeto creado tras crear el proyecto automáticamente en el programa es directamente initialize-water-jug, ya que se presupone que la tarea va a comenzar así, por lo que es automático. Si entramos en la regla se aprecia que el programa ya ha escrito su contenido, que es estándar y solo habrá que completarlo si es necesario (en el caso de water-jug no lo es).

En SOAR hay que hacer reglas para proponer operadores y reglas para aplicar esos operadores. Para aplicar el operador que acabamos de proponer se necesita otra regla. Esta da nombre al estado y crea los "cubos" con el contenido a 0. Esta regla ya añadirá entidades y valores a la Working memory. Vemos que esta regla también ha sido creada por defecto por VisualSoar, pero hay que completarla. El atributo '^empty' no va a ser creado aquí, será creado más tarde y se explicará el porqué.

```
water-jug*apply*initialize-water-jug
```

```
Si el operador 'initialize water-jug' está seleccionado, entonces crea
un cubo de 5l vacío y otro de 3l vacío.

sp {water-jug*apply*initialize-water-jug
   (state <s> ^operator <o>)
   (<o> ^name initialize-water-jug)
-->
   (<s> ^name water-jug
        ^jug <j1>
        ^jug <j2>)
   (<j1> ^volume 5
         ^contents 0)
   (<j2> ^volume 3
         ^contents 0)}
```

VisualSoar, como se menciona previamente, puede hacer debugg del código creado. Hacemos click en *datamap* y en "*check all productions aganist the datamap"* y se generarán varios warnings en azul. Esto ocurre porque se han creado nuevas entidades en las reglas que no están representadas en la working memory. Para arreglar esto abrimos el *datamap* de 'water-jug' y añadir al final del árbol las estructuras nuevas creadas con sus tipos. Hacemos click derecho en 'water-jug' y *add identifier* . Deberemos darle nombre, en este caso queremos crear la entidad 'jug'. Se nos creara una carpeta con ese nombre. Hacemos click derecho sobre ella *Add integer* y le ponemos el nombre de contents, y otra vez para volume y empty. (Se podrían usar otros *Add* para escribir otros tipos de contenido que no fueran integer). Tras realizar esto el programa no debería detectar ningún error adicional. Más adelante podemos crear nosotros los identificadores a mano.

El atributo empty se crea de forma diferente ya que nos interesa que el valor se vaya modificando dinámicamente tras cada cambio, en la elaboración del estado. Esto se hace creando una regla especial que testeará el estado y creara la nueva estructura cada vez. Este tipo de regla se llama 'state elaboration rule'. En este caso cuando el ciclo de SOAR pase por la fase de elaboración del estado se calculará la cantidad de espacio libre en el cubo:

```
water-jug*elaborate*empty
Si el estado se llama water-jug y un cubo tiene v de volumen y
actualmente contiene c litros, añade que tiene v – c espacio libre.

sp {water-jug*elaborate*empty
   (state <s> ^name water-jug
              ^jug <j>)
   (<j> ^volume <v>
        ^contents <c>)
-->
   (<j> ^empty (- <v> <c>))}
```

VisualSoar tiene una carpeta llamada "elaborations" en la "operator window" para guardar los archivos de las "state elaborations". Si se hace click derecho y *Add a File* podremos nombrar el archivo, en este caso 'empty'. Si expandimos la carpeta se puede

ver el archivo recién creado y dos archivos por defecto llamados '_all' y 'top-state'. Estos ficheros son creados por SOAR y son necesarios para trabajar con sub-estados. Se añadirá la regla escrita previamente al nuevo archivo (existen plantillas que pueden ser utilizadas, *insert template*).

Lo siguiente va a ser escribir los operadores seleccionables. *Fill, empty y pour.* Para ello se hace click derecho sobre 'water-jug' *"Add suboperator"*. Se le da nombre y se escribe el código.

```
water-jug*propose*fill
Si el estado se llama water-jug y un cubo tiene espacio, propón llenar
el cubo

sp {water-jug*propose*fill
   (state <s> ^name water-jug
              ^jug <j>)
   (<j> -^empty 0)
-->
   (<s> ^operator <o> + =)
    (<o> ^name fill ^fill-jug <j>)}


water-jug*propose*empty
Si el estado se llama water-jug y un cubo no está vacío, propón vaciar
el cubo

sp {water-jug*propose*empty
   (state <s> ^name water-jug
              ^jug <j>)
   (<j> ^contents > 0)
-->
   (<s> ^operator <o> + =)
    (<o> ^name empty ^empty-jug <j>)}



water-jug*propose*pour
Si el estado se llama water-jug y un cubo tiene espacio y el otro no
está vacío, propón verter el contenido de un cubo a otro.

sp {water-jug*propose*pour
   (state <s> ^name water-jug
              ^jug <j>
              ^jug <i>)
   (<j> -^empty 0)
   (<i> ^contents > 0)
-->
   (<s> ^operator <o> + =)
    (<o> ^name pour ^fill-jug <j> ^empty-jug <i>)}
```

Se necesita que cada operador tenga un atributo adicional, fill-jug o empty-jug, que tienen la misma estructura que el jug previo que hemos hecho, y debido a que puede sufrir modificaciones que queremos que se vean reflejadas vamos a hacer una copia-

link de “jug” dentro de los operadores con los nuevos nombres. Para eso mantenemos clickado con el botón izquierdo la carpeta “jug” y la arrastramos presionando ctrl+shift hasta soltarla encima de la carpeta del operador. Con esto nos aparecerá dentro una carpeta-link con nombre “jug”. Con el botón derecho se puede renombrar para llamarlo fill-jug o empty-jug.
Ahora se deben escribir las reglas de aplicación de los operadores. Serán escritos en el mismo archivo creado para cada una de las propuestas.

```
water-jug*apply*fill
Si la tarea es water-jug y el operador fill esta seleccionado para un
jug, entonces modifica el contenido de jugs a su volumen.

sp {water-jug*apply*fill
   (state <s> ^name water-jug
              ^operator <o>
              ^jug <j>)
   (<o> ^name fill
        ^fill-jug <j>)
   (<j> ^volume <volume>
        ^contents <contents>)
-->
   (<j> ^contents <volume>)
   (<j> ^contents <contents> -)}
```

Para la regla de aplicación de Pour hay dos situaciones: que el cubo a llenar tenga espacio suficiente para toda el agua del otro cubo y la situación de que se tenga que quedar parte en el cubo a vaciar, haremos 2 reglas diferentes para gestionar esto:

```
water-jug*apply*pour*will-empty-empty-jug
Si la tarea es water-jug y pour operator esta seleccionado, y el
contenido del cubo a vaciar es menor o igual al espacio libre del cubo
a llenar, entonces cambia el contenido del cubo vaciado a 0 y el
contenido del cubo a llenar a la suma de los dos cubos.

sp {water-jug*apply*pour*will-empty-empty-jug
     (state <s> ^name water-jug
                ^operator <o>)
     (<o> ^name pour
          ^empty-jug <i>
          ^fill-jug <j>)
     (<j> ^volume <jvol>
          ^contents <jcon>
          ^empty <jempty>)
     (<i> ^volume <ivol>
          ^contents { <icon> <= <jempty> })
-->
     (<i> ^contents 0
          ^contents <icon> -)
     (<j> ^contents (+ <jcon> <icon>)
          ^contents <jcon> -)}


water-jug*apply*pour*will-not-empty-empty-jug
```

```
Si la tarea es water-jug y el operador pour esta seleccionado y el
contenido del cubo a vaciar es mayor que el espacio libro del cubo a
ser llenado, entonces cambia el contenido del cubo a vaciar a su
contenido menos el espacio vacio del cubo a ser llenado y cambia el
contenido del cubo a llenar por su volumen.

sp {water-jug*apply*pour*will-not-empty-empty-jug
   (state <s> ^name water-jug
              ^operator <o>)
   (<o> ^name pour
        ^empty-jug <i>
        ^fill-jug <j>)
   (<i> ^volume <ivol>
        ^contents { <icon> > <jempty> })
   (<j> ^volume <jvol>
        ^contents <jcon>
        ^empty <jempty>)
-->
   (<i> ^contents (- <icon> <jempty>)
        ^contents <icon> -)
   (<j> ^contents <jvol>
        ^contents <jcon> -)}
```

Reglas de monitorización

Para mejorar la lectura de los resultados en SOAR, ya que no tenemos una interfaz más allá del simulador, se hará con unas últimas reglas que escribirán por pantalla el estado de los cubos y las acciones. Estas utilizarán el sistema de escritura por presentado anteriormente. Cuando queramos escribir el texto usaremos "|" y podremos representar el contenido de las variables utilizando la misma forma que en el resto de las reglas, <identificador>. La opción (crlf) significa "carriage-return and linefeed" y ayuda a la correcta visualización del texto. Las reglas son sencillas.

```
sp {water-jug*monitor*state
   (state <s> ^name water-jug
              ^jug <i> <j>)
   (<i> ^volume 3 ^contents <icon>)
   (<j> ^volume 5 ^contents <jcon>)
-->
   (write (crlf) | 3:| <icon> | 5:| <jcon> )}


sp {water-jug*monitor*operator-application*empty
   (state <s> ^name water-jug
              ^operator <o>)
   (<o> ^name empty
        ^empty-jug.volume <volume>)
-->
   (write | EMPTY(| <volume> |)|)}


sp {water-jug*monitor*operator-application*fill
   (state <s> ^name water-jug
              ^operator <o>)
```

```
   (<o> ^name fill
        ^fill-jug.volume <volume>)
-->
   (write | FILL(| <volume> |)|)}

sp {water-jug*monitor*operator-application*pour
   (state <s> ^name water-jug
              ^operator <o>)
   (<o> ^name pour
        ^empty-jug <i>
        ^fill-jug <j>)
   (<i> ^volume <ivol> ^contents <icon>)
   (<j> ^volume <jvol> ^contents <jcon>)
-->
   (write | POUR(| <ivol> |:| <icon> |,| <jvol> |:| <jcon> |)|)}
```

Estado final

Para terminar será necesario poder reconocer el estado final deseado de la tarea. Esto se puede hacer de diferentes formas, aquí se ha elegido una de ellas aunque otras podrían ser válidas. Por un lado tenemos usar una regla básica como las demás que tenga como precondiciones que el cubo de 3l contenga 1l solamente. Sin embargo , si el estado final pudiera ser modificado por algo, por un sensor, por ejemplo, la mejor idea sería codificarlo en la Working memory e ir comprobándolo con ella con otra regla hecha para ello del mismo modo que el parámetro empty se reinicia con cada cambio también se comprobaría si se ha llegado a ese estado. La idea es útil en algunos casos, pero para este problema se va a usar la solución más simple.

```
sp {water-jug*detect*goal*achieved
   (state <s> ^name water-jug
              ^jug <j>)
   (<j> ^volume 3
        ^contents 1)
-->
   (write (crlf) |The problem has been solved.|)
   (halt)}
```

Search control

Finalmente se escribirán unas reglas que, aunque no son necesarias, son recomendables, y mejoran considerablemente la velocidad del simulador para encontrar el resultado óptimo. Estas reglas van a servir para evitar que el simulador elija operadores opuestos uno tras otro. Por ejemplo, llenamos el cubo y posteriormente elegimos vaciarlo, esto tiene poca utilidad y solo va a retrasar llegar al final. Por ello se guarda el operador usado inmediatamente anterior al que vamos a aplicar y si ocurre esta situación se pone el operador en reject (<) para evitar su elección. Esto hay que programarlo, ya que SOAR no lo hace por sí solo.

```
sp {water-jug*record*operator
   (state <s> ^name water-jug
              ^operator.name <name>)
-->
   (<s> ^last-operator <name>)}

sp {water-jug*remove*last-operator
   (state <s> ^name water-jug
              ^last-operator <name>
              ^operator.name <> <name>)
-->
   (<s> ^last-operator <name> -)}

sp {water-jug*select*fill*empty*worst
   (state <s> ^name water-jug
              ^last-operator fill
              ^operator <o> +)
   (<o> ^name empty)
-->
   (<s> ^operator <o> <)}

sp {water-jug*select*empty*fill*worst
   (state <s> ^name water-jug
              ^last-operator empty
              ^operator <o> +)
   (<o> ^name fill)
-->
   (<s> ^operator <o> <)}

sp {water-jug*select*pour*pour*worst
   (state <s> ^name water-jug
              ^last-operator pour
              ^operator <o> +)
   (<o> ^name pour)
-->
   (<s> ^operator <o> <)}

sp {water-jug*select*fill*pour*after*fill-empty
   (state <s> ^name water-jug
              ^last-operator << fill empty >>
              ^operator <o> +)
   (<o> ^name pour)
-->
   (<s> ^operator <o> >)}
```

Análisis de la traza de la ejecución del Agente

Vamos a hacer una traza de ejemplo del problema Water-jug de al menos suficientes pasos para entender su funcionamiento y razonamiento. Para esto usamos las reglas explicadas previamente para crear un programa y lo ejecutamos en el debugger. Iremos pasando por las 5 partes de cada ciclo, input, propose, decision, application y output. Sin embargo debido a la naturaleza del problema no existe input ni output, ya que el sistema no recibe información del exterior ni manda información en ningún

momento. Se van a utilizar diagramas para mostrar el contenido de la Working memory en cada momento tras los cambios significativos. De esta manera podemos ver como evoluciona el estado del agente tras las aplicaciones de las reglas.

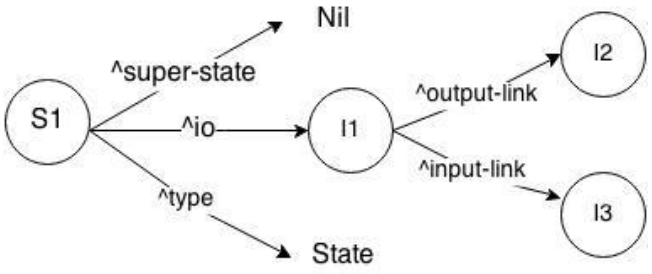


Figura a1.5 *- Diagrama del estado inicial*

Inicialmente el sistema tiene en la Working memory un estado S1 con un atributo super-estado a null, los atributos de entrada y salida y el tipo: estado (ver Figura 1.5). Las reglas creadas para este problema contienen una regla inicial que cuadrará con este estado. Es "propose*initialize*water-jug*" cuyas precondiciones son: *(state <s> ^superstate nil -^name).* Como, efectivamente, super-state es nil y no existe un nombre ésta será la primera regla elegida. Toda la letra en cursiva representa los pasos y decisiones del agente tal como los podemos observar en el debugger.

--- input phase ---
--- propose phase ---
*Firing water-jug*propose*initialize-water-jug*
-->
(O1 ^name initialize-water-jug +)
(S1 ^operator O1 +)
=>WM: (19: S1 ^operator O1 +)
=>WM: (18: O1 ^name initialize-water-jug)

Cuando se va a ejecutar una regla las acciones aparecen por pantalla tal como están escritas en la regla (justo después del →). Posteriormente te muestra los cambios o adiciones a la WM (Working memory). Por lo que en la Working memory nuestro estado quedara con un atributo ^operator O1 + y el atributo ^name del nuevo O1 con el valor "initialize-water-jug" (ver Figura a1.6).

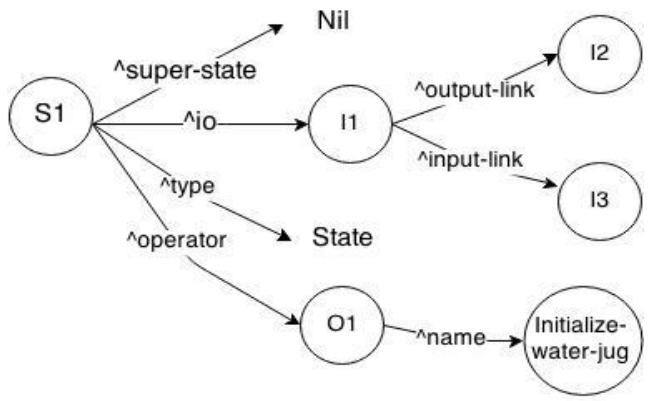


Figura a1.6 *- Diagrama del estado con el primer operador propuesto*

--- decision phase ---
(S1 ^operator O1 +)
=>WM: (20: S1 ^operator O1)
1: O: O1 (initialize-water-jug)

Como en nuestro estado solo tenemos 1 posible operador a aplicar que es aceptable (+) en la decision phase se elegirá y aplicará ese. El operador había sido propuesto por la regla anterior, ahora la regla que se ejecutará será "apply*initialize-water-jug". Para cada operador siempre habrá una regla de propuesta y una de aplicación.

--- apply phase ---
--- Firing Productions (PE) For State At Depth 1 ---
*Firing water-jug*apply*initialize-water-jug*
-->
(J1 ^contents 0 + :O)
(J1 ^volume 5 + :O)
(I4 ^contents 0 + :O)
(I4 ^volume 3 + :O)
(S1 ^jug J1 + :O)
(S1 ^jug I4 + :O)
(S1 ^name water-jug + :O)
--- Change Working Memory (PE) ---
=>WM: (27: J1 ^contents 0)
=>WM: (26: J1 ^volume 5)
=>WM: (25: I4 ^contents 0)
=>WM: (24: I4 ^volume 3)
=>WM: (23: S1 ^jug I4)
=>WM: (22: S1 ^jug J1)
=>WM: (21: S1 ^name water-jug)

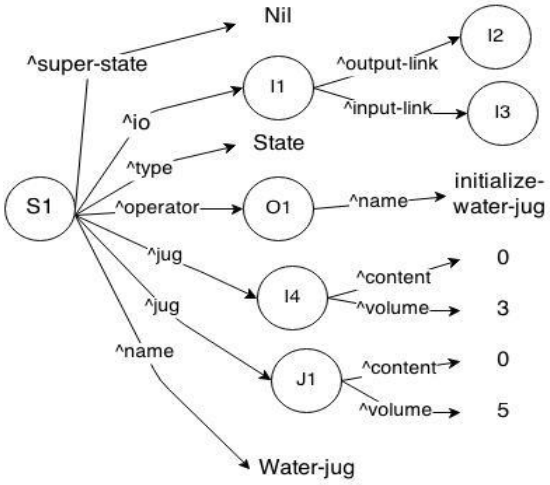


Figura a1.7 – *Diagrama del estado tras aplicar el primer operador*

Esta regla modificará la working memory poniendo los valores iniciales del problema (ver Figura a1.7). Las entidades de los cubos, que en este caso se llamarán I4 y J1 en la Working memory (estos nombres son elegidos por el sistema). Con sus valores iniciales de contenido y volumen tal como indica la regla.

Inmediatamente después de esto con los nuevos valores en el estado van ejecutándose otras reglas. Por un lado la regla de "elaborate*empty" que debe escribir el valor del atributo empty, el cuál como ya se ha explicado se calcula cada vez. Esta regla se va a ejecutar dos veces, ya que existen 2 cubos y hay que calcularlo dos veces. *(<j> ^empty <capacidad> +) (ver Figura a1.8).* Lo siguiente en saltar será el "monitor*state" que hemos programado para que escriba por pantalla y veamos mejor el estado en el que se encuentran los cubos en ese momento, ahora mismo: 5:0,3:0.

--- Firing Productions (IE) For State At Depth 1 ---

*Firing water-jug*elaborate*empty*
-->
(I4 ^empty 3 +)
*Firing water-jug*monitor*state*
-->
5:0 3:0
*Firing water-jug*elaborate*empty*
-->
(J1 ^empty 5 +)
*Retracting water-jug*propose*initialize-water-jug*
-->
(O1 ^name initialize-water-jug +)
(S1 ^operator O1 +)
--- Change Working Memory (IE) ---
=>WM: (29: J1 ^empty 5)
=>WM: (28: I4 ^empty 3)
<=WM: (19: S1 ^operator O1 +)
<=WM: (20: S1 ^operator O1)
<=WM: (18: O1 ^name initialize-water-jug)

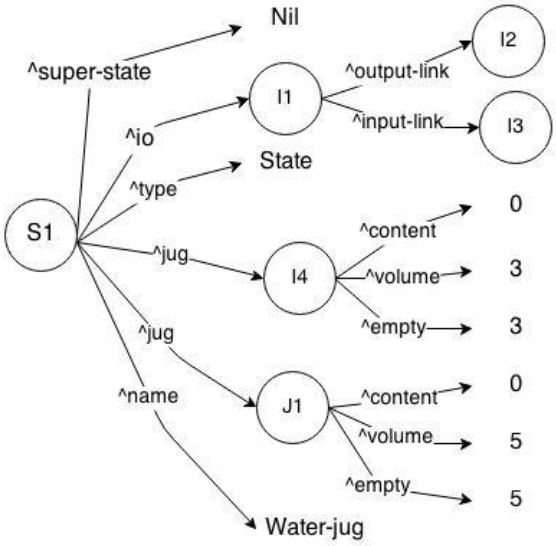


Figura a1.8 – *Diagrama del estado tras aplicar la elaboration "empty"*

Debido a los cambios creados en el estado actual la regla "propose*initialize-water-jug" no va a poder ser cierta y ejecutarse más, por lo que se retracta. Aquí vemos, cuando se modifica la Working memory, datos de entrada, con el símbolo "=>" que son los que vamos a añadir nuevos, como hemos hecho antes y datos que salen de la Working memory, con el símbolo: "<=" y que serán borrados, en este caso porque el operador O1 ya se ha aplicado y no puede volver a ser elegido en la propuesta de operadores.

Tras los cálculos de empty ya se tienen los valores necesarios para que la regla "propose*fill" pueda ser propuesta. En este caso se realizará 2 veces, ya que se realiza una vez para el cubo de 5l y otra para el de 3l. Estas reglas añaden al estado inicial dos posibles operadores nuevos iguales aceptables pero indiferentes, por lo que tienen un 50% de probabilidad de ser elegidos. Se acaba la fase de aplicación y se pasa a la fase de output (ver Figura a1.9).

--- Firing Productions (IE) For State At Depth 1 ---
*Firing water-jug*propose*fill*
-->
(O2 ^fill-jug I4 +)
(O2 ^name fill +)
(S1 ^operator O2 =)
(S1 ^operator O2 +)
*Firing water-jug*propose*fill*
-->
(O3 ^fill-jug J1 +)
(O3 ^name fill +)
(S1 ^operator O3 =)
(S1 ^operator O3 +)
--- Change Working Memory (IE) ---
=>WM: (35: S1 ^operator O3 +)

=>WM: (34: S1 ^operator O2 +)
=>WM: (33: O3 ^fill-jug J1)
=>WM: (32: O3 ^name fill)
=>WM: (31: O2 ^fill-jug I4)
=>WM: (30: O2 ^name fill)
--- output phase ---

Los dos operadores propuestos "cuelgan" del estado inicial, con su nombre y a que cubo afectan. Comienza un nuevo ciclo, sin embargo esta vez ni en input ni en propose phase vamos a tener nada, ya que los operadores ya han sido propuestos anteriormente debido a los cambios en el estado inicial. Vamos al estado de decisión.

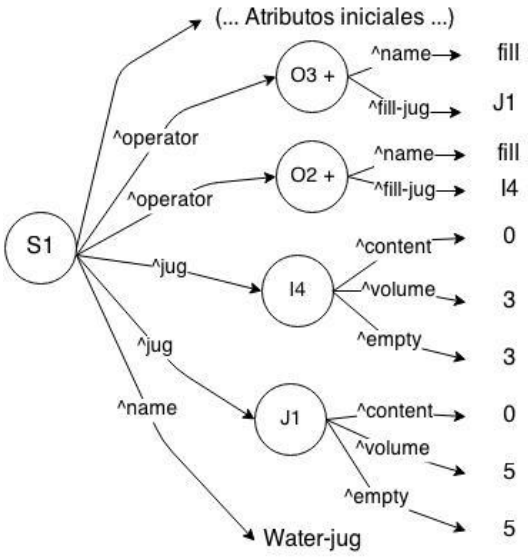


Figura a1.9 –
Diagrama del estado con los 2 operadores fill propuestos

--- input phase ---
--- propose phase ---
--- decision phase ---
(S1 ^operator O2 +)
(S1 ^operator O3 +)
(S1 ^operator O2 =)
(S1 ^operator O3 =)
=>WM: (36: S1 ^operator O3)
2: O: O3 (fill)

En el estado de decisión se decide elegir uno de los dos operadores iguales con la misma probabilidad, en este caso el llenado del cubo de 5l. Se modifica la working memory añadiendo al estado inicial otro atributo ^operator con el valor O3, sin el +, para mostrar que es el que se ha elegido para aplicar. En la fase de aplicación salta la regla de monitorización "water-jug*monitor*operator-application*fill" con lo que nos sale por pantalla el texto que habíamos escrito en la regla "FILL (5)". Salta la regla de aplicación del "fill" "water-jug*apply*fill" eliminando el atributo de contenido 0 para el cubo de 5l y creando uno nuevo con contenido 5l en la working memory.

--- apply phase ---
--- Firing Productions (IE) For State At Depth 1 ---
*Firing water-jug*monitor*operator-application*fill*
21 36 32 33 26

-->
FILL(5)
--- Change Working Memory (IE) ---
--- Firing Productions (PE) For State At Depth 1 ---
Firing water-jug*apply*fill
21 36 32 33 22 26 27
-->
(J1 ^contents 0 - :O)
(J1 ^contents 5 + :O)
--- Change Working Memory (PE) ---
=>WM: (37: J1 ^contents 5)
<=WM: (27: J1 ^contents 0)

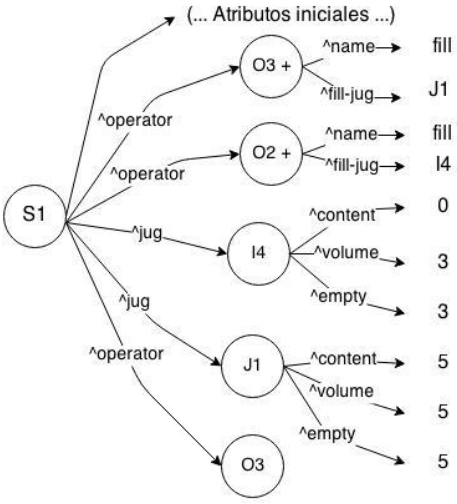


Figura a1.10 – *Diagrama del estado tras aplicar fill*

Con estos cambios otras reglas casan con el estado de la Working memory por lo que comienzan a proponerse operadores (ver Figura a1.10). La regla que aparece como propuesta es "pour". "water-jug*propose*pour" Llenando el cubo de 3l con el contenido que hay ahora en el de 5l, para lo cual se añadirá un operador propuesto O4 + al estado inicial con sus valores y nombre. Justo después salta la regla de monitorización, que nos muestra el estado actual 5:5 3:0. Después se recalcula el atributo empty, ya que el contenido de un cubo se ha modificado, ahora el cubo de 5l debe aparecer como que no tiene más espacio. Tras esto otra regla se puede aplicar, en principio al estado actual, que es "empty" del cubo que acabamos de llenar, se propone. Tras esto se realizan todos los cambios en la Working memory, que veremos reflejados en el diagrama y se retracta el operador "fill" que hemos aplicado, quitando también esos datos del diagrama (ver Figura a1.11).

--- Firing Productions (IE) For State At Depth 1 ---
Firing water-jug*propose*pour
21 23 28 22 37
-->
(O4 ^fill-jug I4 +)
(O4 ^empty-jug J1 +)
(O4 ^name pour +)
(S1 ^operator O4 =)
(S1 ^operator O4 +)
Firing water-jug*monitor*state
21 23 24 25 22 26 37
-->
5:5 3:0
Firing water-jug*elaborate*empty
21 22 26 37
-->
(J1 ^empty 0 +)
Firing water-jug*propose*empty

21 22 37
-->
(O5 ^empty-jug J1 +)
(O5 ^name empty +)
(S1 ^operator O5 =)
(S1 ^operator O5 +)
*Retracting water-jug*elaborate*empty*
21 22 26 27
-->
(J1 ^empty 5 +)
--- Change Working Memory (IE) ---
=>WM: (45: S1 ^operator O5 +)
=>WM: (44: S1 ^operator O4 +)
=>WM: (43: O5 ^empty-jug J1)
=>WM: (42: O5 ^name empty)
=>WM: (41: J1 ^empty 0)
=>WM: (40: O4 ^fill-jug I4)
=>WM: (39: O4 ^empty-jug J1)
=>WM: (38: O4 ^name pour)
<=WM: (29: J1 ^empty 5)
--- Firing Productions (IE) For State At Depth 1 ---
*Retracting water-jug*propose*fill*
21 22 29
-->
(O3 ^fill-jug J1 +)
(O3 ^name fill +)
(S1 ^operator O3 =)
(S1 ^operator O3 +)
--- Change Working Memory (IE) ---
<=WM: (35: S1 ^operator O3 +)
<=WM: (36: S1 ^operator O3)
<=WM: (33: O3 ^fill-jug J1)
<=WM: (32: O3 ^name fill)
--- Firing Productions (IE) For State At Depth 1 ---
--- Change Working Memory (IE) ---
--- output phase ---

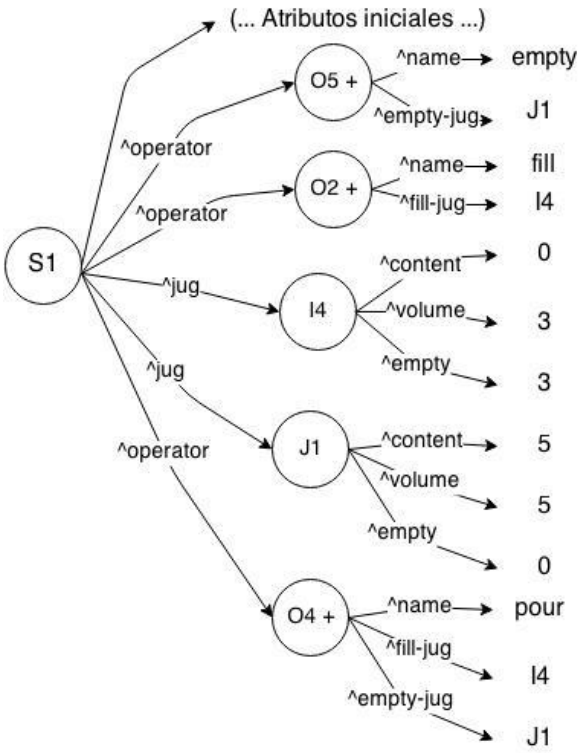


Figura a1.11 *- Diagrama del estado con nuevos operadores propuestos*

El último ciclo que vamos a analizar comienza, de nuevo, en la fase de decisión que en este caso debe elegir entre, "fill" del cubo de 3l, "pour" y "empty" que ya tenemos en nuestro estado. Aleatoriamente ha sido elegido el operador "fill" y debido a ello vamos a ver que el operador pour deberá ser retractado debido a que ya no cumplirá los requisitos para ser elegido. Tenemos el siguiente código:

--- input phase ---
--- propose phase ---
--- decision phase ---
(S1 ^operator O4 +)
(S1 ^operator O5 +)
(S1 ^operator O2 +)
(S1 ^operator O4 =)
(S1 ^operator O5 =)
(S1 ^operator O2 =)
=>WM: (46: S1 ^operator O2)
3: O: O2 (fill)

--- apply phase ---
--- Firing Productions (IE) For State At Depth 1 ---
*Firing water-jug*monitor*operator-application*fill*
21 46 30 31 24
-->
FILL(3)
--- Change Working Memory (IE) ---
--- Firing Productions (PE) For State At Depth 1 ---
*Firing water-jug*apply*fill*
21 46 30 31 23 24 25
-->
(I4 ^contents 0 - :O)
(I4 ^contents 3 + :O)
--- Change Working Memory (PE) ---
=>WM: (47: I4 ^contents 3)
<=WM: (25: I4 ^contents 0)
--- Firing Productions (IE) For State At Depth 1 ---
*Firing water-jug*monitor*state*
21 23 24 47 22 26 37
-->
5:5 3:3
*Firing water-jug*elaborate*empty*
21 23 24 47
-->
(I4 ^empty 0 +)
*Firing water-jug*propose*empty*
21 23 47
-->
(O6 ^empty-jug I4 +)
(O6 ^name empty +)
(S1 ^operator O6 =)
(S1 ^operator O6 +)
*Retracting water-jug*elaborate*empty*
21 23 24 25
-->
(I4 ^empty 3 +)
--- Change Working Memory (IE) ---
=>WM: (51: S1 ^operator O6 +)
=>WM: (50: O6 ^empty-jug I4)
=>WM: (49: O6 ^name empty)
=>WM: (48: I4 ^empty 0)
<=WM: (28: I4 ^empty 3)
--- Firing Productions (IE) For State At Depth 1 ---
*Retracting water-jug*propose*pour*
21 23 28 22 37
-->
(O4 ^fill-jug I4 +)
(O4 ^empty-jug J1 +)
(O4 ^name pour +)
(S1 ^operator O4 =)
(S1 ^operator O4 +)
*Retracting water-jug*propose*fill*
21 23 28
-->
(O2 ^fill-jug I4 +)
(O2 ^name fill +)
(S1 ^operator O2 =)

(S1 ^operator O2 +)
--- Change Working Memory (IE) ---
<=WM: (34: S1 ^operator O2 +)
<=WM: (46: S1 ^operator O2)
<=WM: (44: S1 ^operator O4 +)
<=WM: (40: O4 ^fill-jug I4)
<=WM: (39: O4 ^empty-jug J1)
<=WM: (38: O4 ^name pour)
<=WM: (31: O2 ^fill-jug I4)
<=WM: (30: O2 ^name fill)
--- Firing Productions (IE) For State At Depth 1 ---
--- Change Working Memory (IE) ---
--- output phase ---

Si nos fijamos en las modificaciones a la working memory, nuestro estado final quedará así: (ver Figura a1.12)

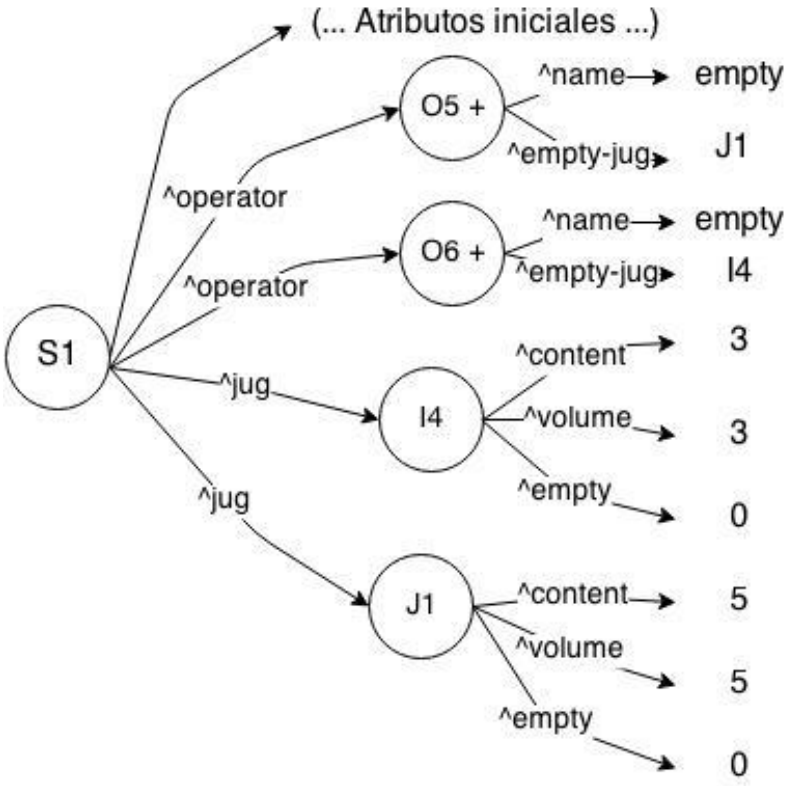


Figura a1.12 – *Diagrama del estado final*

Se puede entender fácilmente el funcionamiento posterior del agente en cada ciclo, que con sus decisiones aleatorias acabará llegando a un estado en el cual el contenido de I4 sea 1, lo que hará saltar la regla final que parará el programa.

La cantidad de ciclos necesarios para llegar a una solución varía notablemente en cada ejecución por ser las decisiones aleatorias, sin embargo existen métodos para disminuir, de media, el número de ciclos necesarios, por ejemplo con el Chunking.

Anexo B

Técnica de aprendizaje: Chunking

SOAR fue diseñado originariamente para ser un solucionador de problemas general, sin aprendizaje. Sin embargo su forma de resolver los problemas y su estructura de memoria soportan el aprendizaje de distintas maneras. La estructura de SOAR determina cuando debe ser creado nuevo conocimiento, qué conocimiento debe ser y cuándo debe ser adquirido.

Cuándo es necesario tener nuevos conocimientos: Los impasses ocurren si y solo si el conocimiento disponible directamente es incompleto o inconsistente. Por lo tanto los impasses indican cuando debe adquirir nuevos conocimientos el sistema.

Qué conocimiento adquirir: mientras resolvemos los problemas dentro de una sub-meta SOAR puede descubrir información que resuelva un impasse. Esta información, si es recordada, puede evitar impasses similares en el futuro.

Cuándo puede adquirirse ese nuevo conocimiento: Cuando una sub-meta es completada, porque un impasse habrá sido resuelto, es cuando podemos añadir ese conocimiento que no era explícitamente conocido de antemano.

El sistema de memoria a largo plazo basado en reglas (producciones) soporta el aprendizaje, por un lado integrar nuevos conocimientos, ya que las producciones son una representación modular de éstos, por lo que solo se requiere añadir una nueva producción, sin importar lo que hubiera en la memoria previamente. Y por otro lado usar esos nuevos conocimientos inmediatamente, sin precisar una sincronización.

Para la construcción de un "chunk", una regla nueva, las condiciones consistirán en aquellos aspectos de la situación que existían previamente en ese punto de la ejecución, y que son examinados en el proceso, mientras que las acciones de la regla nueva consistirán en el resultado de esa parte del problema.

Cuando llegamos al final de una sub-meta del problema los elementos de la Working memory son convertidos en condiciones y acciones de una o más producciones, tenemos 3 pasos clave en la creación del "chunk", buscar y guardar todas las condiciones y acciones, convertir en variables los identificadores y optimizarlo.

El comportamiento por defecto en SOAR es crear un "chunk" siempre en cada resolución de un impasse, cada vez que una sub-meta sea acabada. Otra opción sería usar "botton-up", un sistema diferente, para ir creando los "chunks" una vez los impasses vayan enlazando sus terminaciones, guardando toda la información para crear un "chunk" menos específico pero más potente. Sin embargo no está claro que

ninguna de las dos opciones sea más eficiente que la otra, por lo que están implementadas ambas en la arquitectura.

Recogida de condiciones y acciones

Para la creación del "chunk" se comprueban aquellos aspectos de la situación (entidades de la Working memory) que existían previamente al comienzo de la "meta" y que puedan ser relevantes para llegar al resultado final, en relación con los resultados obtenidos. Estas entidades son las que tenían coincidencias con las reglas que se han ido ejecutando en el proceso de esta "meta" (o de las "sub-metas") pero que ya existían previamente a empezar el proceso. Estos son los elementos que se consideran implícitamente relevantes para llegar al final. En la "referenced-list" SOAR va guardando para cada "meta" activa estos elementos. Permite que cualquier producción que se ejecute en una "meta" concreta de la ejecución pueda ser añadida a la lista, pero siempre en la zona que le corresponda, lo cual afectará a los "chunks" creados para ésta y todas las "sub-metas" dependientes de ésta, sin embargo no afecta a los "chunks" creados previamente en estas "sub-metas" ya que se han creado para cierto entorno concreto, aunque luego cambie. Si ha habido datos de entrada desde el exterior o una "sub-meta" se ha resuelto con un conocimiento independiente del dominio del problema no se crean reglas nuevas.

Las acciones de un "chunk" están basadas en los resultados de la "sub-meta" en la cual se ha creado. Si no hay resultados no puede crearse la regla.

Ejemplo: Tenemos un tablero de 3x3 con 8 fichas numeradas que no se pueden levantar del tablero y un hueco en medio. Debemos ir arrastrando las fichas por el tablero utilizando el hueco disponible para ir cambiándolas de posición hasta conseguir una disposición final concreta. Nuestro estado inicial y deseado son los siguientes (/ marca el espacio vacío):

2 3 1	1 2 3
/ 8 4	8 / 4
7 6 5	7 6 5

Nuestra traza del problema es la siguiente:

```
1 G1 solucionar-8puzle
2 P1 8puzle-sd
3 S1
        2 3 1
        / 8 4
        7 6 5
4 01 espacio-vacío
5 ➔G2 (resolución-no-change)
6   P2 8puzle
7   S1
8   ➔G3 (resolución-tie operador)
9     P3 tie
10    S2 {iz, arriba, abajo}
11    05 evalua-objeto{02(iz)}
```

```
12      ➔G4 (resolución-no-change)
13          P2 8puzle
14          S1
15          02 iz
16          S3
            2 3 1
            8 / 4
            7 6 5
17          02 iz
18           S4
19  S4
20  08 espacio-1
```

Lo primero es inicializar el problema, con la "meta" G1 – 8puzle. G1 es el identificador asignado por SOAR, después se cargan los datos de "problem space" P1 – 8puzle-sd, que tienen la información sobre qué entidades están involucradas y deben ser tomadas en cuenta, y algunas configuraciones. S1 es seleccionado con el estado inicial y los operadores de ordenación aparecen, siendo elegido el 01 espacio-vacío. Como el operador no está implementado como una regla saltará un impasse G2 del tipo no-change para completar la "sub-meta" de implementar el operador espacio-vacío, el cual se completará en el momento en el que éste espacio vacío esté en el lugar deseado según su posicionamiento final. En este caso en medio del tablero, en vez de en un lado. Para que sea posible finalizar la "sub-meta" el "problem-space" en el que nos debemos poner es el estado final y así realizar una comparación entre el estado en el que nos encontramos y el final , sabiendo que pieza vamos a mover, para llegar hasta ese estado.

Por lo tanto se selecciona el estado inicial de nuevo, que genera un tie-impasse, ya que para mover de lugar el espacio-vacío se pueden realizar 3 operadores, bajar el 2, subir el 7 y mover el 8 a la izquierda. El tie impasse generado (G3) genera también un tie problem-space que copiará todas las configuraciones previas. Los operadores disponibles, que tienen ciertas preferencias entran en la posible elección. Se elige el operador 05, que mueve el número 8 a la izquierda. Debido a que no hay ninguna producción que sea capaz directamente de evaluar la posición nueva de la ficha 8 se entra en un nuevo impasse no-change G4 para el estado S1. Vemos que las decisiones tomadas nos han llevado hasta el estado final S3, por lo que se crean las preferencias por las que desde el estado S1 se debe tomar el operador O5 para llegar al estado S3, sin necesidad de pasar por G3 y G4. Si con 05 no se hubiera llegado al estado final se hubiese generado otro tie impasse con un nuevo problema-space y así la combinación de tie impasse con no-change impase generarían una búsqueda en profundidad hasta encontrar los resultados.

Para este ejemplo consideramos la "sub-meta" G4, ésta ha sido creada como resultado de un impasse no-change para evaluar el operador que movería la ficha 8 a la izquierda. G4 tiene como entidades que salen de sí a D1, el estado deseado con respecto al espacio vacío, P2 que es el "problem-space" del 8puzle, S2 que es el estado

sobre el cual se va a aplicar el operador, y el operador a ser evaluado O2. Para añadir todo esto a G4 se ha accedido desde el operador a ser evaluado O5, por lo que todas las referencias hechas se añaden a la "list-reference" de la que hemos hablado antes. Se añade: G3 apunta a O5, O5 es el objeto a evaluar, O5 apunta a P2 problem-space, S1 estado inicial, O2 operador sobre 8, D1 estado deseado.

Una vez el operador es aplicado, S1 se convierte en S3, el cual concuerda con D1. La forma en la que se representan internamente tanto S3 como D1 genera también añadidos a la "list-reference" ya que se han tenido que hacer comprobaciones en las relaciones entre las fichas y sus posiciones para comprobar que eran similares.

Previamente a comenzar la "sub-meta" en el G3 debido a que íbamos a evaluar un objeto se había creado una entidad E1 que valoraría el acierto o fallo de la evaluación. Tras la aplicación del operador en G4 hemos visto que ha sido un acierto, y el valor de"acierto" es generado para E1, como E1 existía previamente a la "sub-meta" se convierte en la acción del chunk que será creado, con las condiciones incluidas en la "list-reference" que tenían que ver con O5, y no con G4, ya que la "sub-meta" G4 no existía previamente y no será generada , ya que el "chunk" que estamos creando lo evitará, generado un estado de acierto en la evaluación izquierda en esta situación, sin entrar en un impasse de no-change.

Crear variables de los identificadores

Cuando las condiciones y las acciones han sido determinadas, todos los identificadores son remplazados por variables, mientras que las constantes como "evaluate-object" o "8puzle" se dejan tal cual. Con el identificador podemos referenciar una instancia de un objeto en la Working memory. Su funcionamiento es a corto plazo, ya que solo dura instanciado mientras el objeto exista en la Working memory. Cada vez que un objeto reaparece en la working memory será referenciado con un identificador diferente. Por lo tanto hay que generalizar en los "chunks", para que puedan adaptarse a los nombres de los identificadores en cualquier situación. Todas las ocurrencias del mismo identificador se reemplazan por la misma variable. SOAR debe suponer qué identificadores son iguales entre si y cuales difieren, basándose solo en la información de la Working memory.

Optimización de "chunks"

Hay tres procesos adicionales que se realizan para optimizar los "chunks", el primero elimina las condiciones que no suponen restricciones para su aplicación a la Working memory. Cada condición de la regla debería ir cerrando el círculo de su posible aplicación sobre una situación en particular, si una condición no supone una restricción no es necesaria ni útil. Eliminar todas estas condiciones no hace a la nueva regla ser más general, sino que elimina carga innecesaria sin afectar a su aplicación.

La segunda optimización es eliminar posibles aplicaciones que generen copias de grupos de entidades de un objeto existente a uno nuevo. Ya que una estrategia común para implementar los operadores es las "sub-metas" es crear un nuevo estado que contenga toda la nueva información y los cambios y acceder al estado anterior con punteros. Si, como es usual, un grupo de ítems similares son copiados al final solo se diferencian por el nombre de las variables, por lo que cada grupo de entidades corresponde con las condiciones de las reglas de forma independiente, generando un numero combinatorio de aplicaciones de la regla. Este problema se resuelve juntando las condiciones de copia similares en una única. Las copias podrán ser generadas igual, pero de forma controlada. Sin embargo con las copias en las producciones de las reglas a veces tenemos problemas, ya que las reglas pueden tener producciones que afecten a una sola entidad, sin copias ni múltiples instanciaciones, por lo que la única solución es dividir en 2 las reglas, una parte con las producciones de las copias y otra con lo demás.

El proceso final de optimización consiste en aplicar una reordenación de las condiciones con un algoritmo, para las producciones del "chunk". La eficiencia de SOAR es sensible a la ordenación de las condiciones de las reglas. Sabiendo cómo es la estructura de la Working memory, se ha creado un algoritmo estático que incrementa significativamente la eficiencia de las coincidencias.

Modificaciones para su uso

Problema de Water-jug

Vamos a modificar el problema "Water-jug" creado en el Anexo 1, para añadirle la posibilidad de usar el Chunking.

Como hemos introducido anteriormente SOAR utiliza la técnica o método de Chunking para su aprendizaje a corto plazo. Automáticamente crea una sub-meta en cualquier caso en el que las preferencias dadas son insuficientes para seleccionar un operador por el procedimiento de decisión. Esto es una señal de que no hay conocimiento suficiente para decidir y es necesario algo más para resolver el problema. El proceso de decisión puede escoger un operador si existe un claro ganador (un operador que domine a los otros basándose en las preferencias disponibles), o cuando hay múltiples operadores indiferentes. Por lo tanto lo primero que vamos a hacer es eliminar las preferencias indiferentes de las propuestas de las reglas para los operadores, que era lo que teníamos hasta ahora en el problema de "Water-jug" y que lo hacía hacer elecciones aleatorias.

Esta es una de las reglas principales:

```
sp {water-jug*propose*fill
```

```
    (state <s> ^name water-jug
    ^jug <j>)
    (<j> ^empty > 0)
-->
    (<s> ^operator <o> += +)
    (<o> ^name fill
    ^jug <j>)}
```

Una vez la preferencia indiferente es eliminada, se vuelve a ejecutar el programa y dónde antes se hacía la primera decisión del programa ahora se generan sub-estados para todas las decisiones donde las preferencias son insuficientes para elegir un único operador por encima de los demás. Este es el tie-impasse. Cuando esto ocurre se crea un sub-estado de manera muy similar al creado en un operator-no-change-impasse. La diferencia principal es que operator-tie tiene los atributos ^elección multiple, ^impasse tie, y ^ítem para cada uno de los operadores del impasse, lo veremos muy claro en el ejemplo más adelante.

En el operator-no-change-impasse la meta era aplicar un operador seleccionado aplicando los operadores hasta encontrar uno que modificara el súper-estado directa o indirectamente añadiendo o quitando atributos o realizando acciones que crearan cambios en sensores implicados en el estado.

En el tie-impasse la meta es determinar qué operador de la tarea es el mejor para aplicar al estado concreto de ese momento, esto se consigue seleccionando y aplicando operadores de evaluación en el sub-estado, uno por cada operador de la tarea. El propósito de estos operadores es crear una evaluación de los otros, como “fallo”, “acierto” o evaluación numérica de como de bien ha funcionado o se ha acercado a su meta final. Estas evaluaciones serán convertidas en preferencias para los operadores de la tarea. Una vez haya suficientes preferencias que hayan sido creadas para seleccionar el operador será seleccionado y el sub-estado se eliminará automáticamente de la Working memory. Es posible que solo sea necesario hacer la evaluación de un pequeño grupo de operadores para llegar a una conclusión y terminar el impasse, por ejemplo si un operador es evaluado y se determina que da resultado directo en la meta de la tarea, esto debería ser suficiente para resolver el impasse.

La forma de evaluar qué operadores pueden ser viables es simular su resultado de aplicación a copias internas del estado en el que ha ocurrido el impasse y evaluar los estados a los que se llegan o crean. La evaluación de estos estados puede basarse en cuál es el estado deseado, cuál sería un estado fallido o en el estado inicial. Muchos problemas pueden evaluarse a sí mismos de alguna manera para estimar lo lejos o cerca que se encuentran de su meta, por ejemplo en el ajedrez a través de números asignados a diferentes piezas en comparación con las de tu adversario.

Aunque encontrar unas buenas funciones de evaluación del estado para un problema específico puede ser complejo, la aproximación general es sencilla y requiere muy poca información del dominio especifico de la tarea. Por ello existe un set de operadores genéricos de SOAR (reglas) que manejan la evaluación y comparación de operadores usando esa aproximación. Este set de reglas genéricas, en una amplia variedad de problemas para planificación simple, pueden ser usadas y están disponibles como parte de la librería, son reglas que vienen de serie en el archivo Agents/default/selection.soar.

Vamos a ver ahora el sistema que tiene la arquitectura para enfrentarse a decisiones. Esta será la base del Chunking, pero no es parte de él, ya que SOAR lo hace aunque tenga desactivado el aprendizaje. El agente entra en una **simulación** con "sub-metas" y "sub-estados" y cuando llega a la solución es cuando se aplica el aprendizaje.

Representación del estado de selección

Si la evaluación de los operadores se basa exclusivamente en un atributo (conseguido, fallo) se puede codificar con solo un atributo al estado ^evaluation success, pero normalmente necesitaremos más información. Esta evaluación debe incluir el operador al cual se refiere la evaluación, además encontraremos útil tener diferentes tipos de evaluaciones para comparar de distintas maneras, como simbólicas (conseguido, fallo) o numéricas (en un rango). Encontraremos útil crear una estructura de evaluación antes de que sea conocida la respuesta por lo que crearemos unos atributos del "objeto" de evaluación que demuestre que se ha creado un valor. Seguirá la siguiente estructura:

- **operator <o>** el identificador del operador a ser evaluado
- **symbolic-value** valores: (success/partial-success/partial-failure/failure/indifferent)
- **numeric-value** [numero]
- **value** a "true" indica que hay un valor numérico o simbólico.
- **desired <d>** el identificador del estado deseado para la evaluación

Creación del estado de selección

El estado de selección es creado automáticamente en respuesta a un impasse. Además los objetos de evaluación serán creados por el ciclo: "operator application". La regla de elaboración del estado inicial necesaria debe ser una que se llame estado de selección. Esta regla no es realmente necesaria pero nos permitirá chequear el nombre del estado en vez del tipo de impasse en todas las reglas restantes.

```
sp {default*selection*elaborate*name
```

```
:default
   (state <s> ^type tie)
-->
   (<s> ^name selection)}
```

Proposición de operadores en selección

El único operador que se puede proponer en el estado de selección es "evaluar operador". Esto debe ser propuesto si hay algún ítem que todavía no tenga un valor de evaluación. Este operador creara primero una evaluación y después hará el cálculo de su valor.

*Selection*propose*evaluate-operator*
Si el estado se llama selection y hay un ítem que no ha sido evaluado con un valor, entonces propón el evaluar operador para ese ítem.

```
sp {selection*propose*evaluate-operator
:default
   (state <s> ^name selection
              ^item <i>)
 -{(state <s> ^evaluation <e>)
   (<e> ^operator <i>
        ^value true)}
-->
   (<s> ^operator <o> +, =)
   (<o> ^name evaluate-operator
        ^operator <i>)}
```

Sin embargo habrá algunos casos donde parte del conocimiento sobre la tarea puede ser usado para la evaluación de los operadores. Por ejemplo en el ajedrez muchos programas usan profundización iterativa para evaluarse. Además veremos un ejemplo donde es muy útil añadir conocimiento de control de búsqueda para el evaluar operador en la selección.

Aplicación de operadores en selección

La aplicación de la evaluación de operadores tiene dos partes. La primera es la creación de la estructura de evaluación que hemos comentado, sin ningún valor. En paralelo con la creación de la estructura el operador se crea con estructuras adicionales para una aplicación más simple. Al final tenemos lo siguiente:

```
(<s> ^evaluation <e>
     ^operator <o>)
(<e> ^superoperator <so>
     ^desired <d>)
(<o> ^name evaluate-operator
     ^superoperator <so>
     ^evaluation <e>
     ^superstate <ss>
     ^superproblem-space <sp>)
```

El atributo "desired" (deseado) es un objeto que describe el estado deseado de la tarea original (la meta). En el problema wáter-jug es una entidad que describe el cubo de tres litros conteniendo solo un litro. La estructura de "desired" está incluida siempre, por lo que la evaluación de un operador puede ser basada en cuánto ayuda a conseguir este estado. Puede ser: alcanzando el estado deseado o basado en un cálculo heurística de la distancia del estado deseado al estado creado por la aplicación del operador a evaluar. El atributo "superproblem-space" es el objeto del problem-space en el súper estado (el anterior a los sub-estados, el inicial). Es una representación explicita en el estado de las propiedades del problem space concreto y puede ser usado para marcar la aplicación de reglas relevantes al problem space actual. El nombre del atributo del estado ha servido para este propósito en sistemas previos, sin embargo es insuficiente generalmente porque contiene solo un símbolo único, el nombre. Si tuviéramos una entidad en vez de solamente un nombre, algunas propiedades adicionales podrían ser incluidas. Esto es importante para la planificación porque la búsqueda "look-ahead" debe usar el mismo problem space y debe copiar el estado actual, lo cual requiere información sobre la estructura del problema.

La segunda parte es el cálculo de la evaluación, que no puede ser hecha directamente con reglas pero requiere su propio sub-estado. Este sub-estado es llamado en la evaluación. El estado original de la tarea es copiado al sub-estado de evaluación. Entonces el operador de la tarea a ser evaluado se aplica al estado. Si el nuevo estado creado puede ser evaluado se devuelve (el resultado) y añadido al objeto de evaluación. Si no es posible su evaluación entonces el estado normal de resolución del problema continua hasta que la evaluación puede ser hecha. Esto puede llevar a más impasses o sub-estados.

Copias del estado inicial

El estado inicial de la simulación debe ser una copia del estado en el cual se ha producido el impasse. Copiar todos los atributos se puede hacer con unas pocas reglas en el problema de Water-jug. Sin embargo, esto significa que con cada nueva tarea nuevas copias del estado deben ser escritas. Para evitar esto se han escrito unas reglas generales que pueden copiar los atributos. Estas son:

- No copiar el estado por defecto, no hacerlo automáticamente
- Atributos de un nivel: copia entidades del estado y preserva sus valores
- Atributos de dos niveles: copia las entidades del estado y crea nuevos identificadores para los valores. Los identificadores compartidos quedan igual.
- Todos los atributos de nivel 1
- Todos los atributos de nivel 2
- No copiar el atributo

- No copiar nada

Si no hay ningún atributo relativo a las copias incluido, el usado por defecto es todos los atributos de nivel 1. El estado deseado también es copiado, basándose en los comandos de copia del estado. Etas reglas soportan 2 niveles de copia. En Water-jug el estado tiene 2 niveles de estructura: los cubos y su contenido, volumen y llenado:

```
(s1 ^jug j1 j2)
(j1 ^volume 3
    ^contents 0
    ^empty 3)
(j2 ^volume 5
    ^contents 0
    ^empty 5)
```

En nuestro problema de Water-jug vemos que si usamos el sistema de copia por defecto las reglas que modifican atributos o entidades de segundo nivel, como el llenado de los cubos modificarían el súper-estado, lo cual es inaceptable, así que es necesario incluir un atributo al estado que diga que se copia hasta nivel 2. El código quedará así:

```
sp {water-jug*elaborate*problem-space
   (state <s> ^name water-jug)
-->
(<s> ^problem-space <p>)
(<p> ^name water-jug
     ^default-state-copy yes
     ^two-level-attributes jug)}
```

Una segunda forma de hacerlo es cambiar la aplicación de los operadores de forma que no modifique las entidades de los objetos del cubo sino que cree nuevos objetos. Por ejemplo nuestra versión de “fill” sería:

```
sp {water-jug*apply*fill
(state <s> ^operator <o>
           ^jug <i>)
(<o> ^name fill
     ^jug <i>)
(<i> ^volume <volume>
     ^contents 0
     ^empty <volume>)
-->
(<s> ^jug <i> -
     ^jug <ni>)
(<ni> ^volume <volume>
      ^contents <volume>
      ^empty 0)}
```

La desventaja de esto es que requiere más cambios a la Working memory cuando se aplica un operador, aunque requiere que menos elementos sean copiados en la creación del estado inicial. Esta aproximación es menos natural ya que implica que un

nuevo cubo es creado en oposición a modificar simplemente su contenido. Es preferible usar la copia de nivel 2.

Selección del operador que debe ser evaluado

Una vez el estado inicial del evaluador de operadores es creado, una copia del operador a ser evaluado debe ser seleccionada. La razón de hacer una copia es que se deben hacer ciertas modificaciones en los atributos, y se deben copiar también los atributos del operador del estado desde el que se llama, como por ejemplo esta línea `(<o> ^name fill ^jug j1)`.

Una vez está hecha la copia, las demás reglas no se aplican a ninguno de los otros operadores propuestos, por lo que la copia creada será siempre seleccionada.

Aplicar el operador seleccionado

Una vez el operador es seleccionado las reglas para aplicarlo se ejecutarán modificando el estado copiado. No hace falta añadir nada ni modificar nada. Para tareas donde el operador original hace operaciones en el mundo externo habrá que escribir reglas para simular el efecto de la selección del operador en ese estado. Por ejemplo un simulador de movimiento.

Evaluación del resultado

Una vez se ha creado un nuevo estado la evaluación puede ser hecha. Se crea una nueva entidad en paralelo a la aplicación del operador: `^tried-tied-operator <o>`.
Esta nueva entidad puede ser testeada por reglas para asegurarse de que se ha hecho bien.
Lo más simple es evaluar si se ha acertado o fallado. Acertado es cuando la meta se consigue. Ya hemos escrito la regla que detecta esto para Water-jug, sin embargo esto solo vale para el final. Esta regla puede ser modificada para crear un elemento de la Working memory que se use como evaluación. La regla de evaluación será la siguiente:

```
sp {water-jug*evaluate*state*success
   (state <s> ^desired <d>
              ^problem-space.name water-jug
              ^jug <j>)
   (<d> ^jug <dj>)
   (<dj> ^volume <v> ^contents <c>)
   (<j> ^volume <v> ^contents <c>)
-->
```

```
   (<s> ^success <d>)}
```

Las preferencias que pueden crearse son: success, partial-success, indifferent, failure y partial-failure. Habrá que incluir estas dos nuevas reglas para que el sistema empiece a funcionar como una búsqueda look-ahead:

```
water-jug*elaborate*problem-space, water-
jug*evaluate*state*success
```

Existen 2 problemas, uno que cuando se llega a una solución en un sub-estado de la pila e sistema debe volver poco a poco sobre sus pasos redescubriendo la solución, esto lo agiliza y utiliza el Chunking, el otro problema es que el sistema normalmente vuelve al mismo estado y genera el mismo problema. Esto se soluciona evaluando los estados que ya hay en la pila para evitar repeticiones y bucles. (Si esto funciona bien podemos evitarnos las reglas que recordaban el operador anterior usado.)

Qué es necesario testear:

1- El estado deseado, que es usado en: (<s1> ^desired <d>)
2- El contenido del estado
3- Que el estado exista después de que el operador haya sido aplicado para su evaluación: (<s1> ^tried-tied-operator)
4- Que haya un duplicado anterior en la pila. No hay nada inherente de soar que controle esto. Hay que añadir una regla que creen cada estado con todos sus súper-estados, y los de los demás.

```
sp {Impasse__Operator_Tie*elaborate*superstate-set
   (state <s> ^superstate <ss>)
-->
(<s> ^superstate-set <ss>)}

sp {Impasse__Operator_Tie*elaborate*superstate-set2
   (state <s> ^superstate.superstate-set <ss>)
-->
(<s> ^superstate-set <ss>)}
```

La regla que detecta los fallos es la siguiente:

```
sp {water-jug*evaluate*state*failure*duplicate
   (state <s2> ^name water-jug
               ^superstate-set <s1>
               ^jug <i1>
               ^jug <i2>
               ^tried-tied-operator)
   (<i1> ^volume 5 ^contents <c1>)
   (<i2> ^volume 3 ^contents <c2>)
   (<s1> ^name water-jug
         ^desired <d>
         ^jug <j1>
         ^jug <j2>)
    (<j1> ^volume 5 ^contents <c1>)
    (<j2> ^volume 3 ^contents <c2>)
```

```
-->
(<s2> ^failure <d>)}
```

Añadiendo esta regla la resolución del problema será más directa, pero usara un exceso de decisiones para solucionarlo. El problema es que tan pronto se produce el resultado, se olvida la solución, por lo que se repite mucho. Es necesario el aprendizaje del que hablamos.

Chunking

El Chunking es invocado cuando se produce un resultado final satisfactorio en un sub-estado. El resultado será la acción de una nueva regla (el operador). Este sistema examina la Working memory para seleccionar los elementos utilizados para crear ese resultado, si alguno de esos elementos están enlazados al super-estado se convierten en pre-condiciones de la regla. Para cada elemento (entidad) el Chunking encuentra la regla que lo ha creado y hace una búsqueda de todo el árbol de reglas creado por los impasses y decisiones tomadas para crear la nueva regla.

Comentario sobre la traza usando Chunking

El problema water-jug utilizando el aprendizaje de Chunking es notablemente más complejo que el explicado anteriormente, ya que los impasses que se generan, duplicando los estados para hacer las evaluaciones de los operadores y las elecciones de éstos, multiplican el número de entidades y sus atributos dificultando mucho su representación con esquemas sencillos como los mostrados en el Anexo 1.

El funcionamiento del agente para resolver el problema con utilización de esta técnica sigue teniendo un componente de aleatoriedad en ciertas decisiones de aplicación de operadores, por lo que de una simulación a otra hay diferencias. Sin embargo el funcionamiento siempre es igual y la traza suele estar en torno a las 21000 líneas teniendo en cuenta que se muestra la mayor cantidad posible de datos para su comprensión y seguimiento (Este dato no es interesante por sí mismo, ya que no cuantifica el sistema objetivamente, sin embargo sí que se puede usar para compararse con sí mismo en otras circunstancias). Lo interesante de este dato, es que una vez el agente ha resuelto el problema una vez utilizando Chunking , si se guardan las reglas creadas por el agente y se reinicia para que empiece de cero pero con lo aprendido se observa un cambio muy significativo, ya que la traza devuelta en esta ocasión son apenas 700 líneas, y si nos fijamos en los resultados el agente prácticamente llega a la solución directamente, sin entrar en impasses y apenas haciendo 2 o 3 movimientos no útiles para llegar a la solución. Con esto se aprecia lo rápido que puede aprender el agente y los buenos resultados.

Existen, en el debugger, las opciones de mostrar cuándo se van creando las reglas nuevas e incluso ver cómo son internamente, cómo las ha construido el sistema. Podemos modificar pocas cosas del Chunking como meros programadores, sin embargo debido a que es totalmente óptimo no ha sido necesario hacer ninguna configuración adicional. Nos hemos encontrado, de todas formas, con un problema que ha habido que solventar sobre la terminación del agente y está expuesto en el apartado de Problemas y Soluciones.

Anexo C

Técnica de aprendizaje: Reinforcement Learning

Los operadores que hasta ahora habíamos tenido en cuenta para la toma de decisiones de los programas en Soar eran los siguientes: Aceptable (+): solo los operadores aceptables son considerados para ser elegidos para aplicarse. Su funcionamiento es diferente dependiendo de si además también son considerados indiferentes o no. Las preferencias diferenciadas (> o <) establecen un relativo "orden" por el cual Soar es capaz de elegir los operadores preferidos en cada decisión. Si las preferencias también tienen la simbología de indiferente (=), entonces una elección aleatoria hará el trabajo de selección, sin embargo existen ciertos parámetros numéricos que pueden afectar a que en esa decisión ciertas opciones tengan más peso (más posibilidades de ser elegidas) afectando a la aleatoriedad.

El reinforcement learning permite al agente alterar su comportamiento en el tiempo modificando dinámicamente las preferencias numéricas de las decisiones indiferentes como respuesta a un sistema de recompensas. Este sistema es contrario al Chunking. Mientras que el Chunking aprende de una vez incrementando su conocimiento para tomar decisiones ante una sub-meta, el Reinforcement funciona de forma incremental y es un tipo de aprendizaje que altera el comportamiento del agente de forma probabilística.

SOAR soporta 2 tipos de exploración, épsilon-greedy y Boltzman(también llamado softmax).
Si usamos épsilon greedy con probabilidad Ɛ (épsilon): el agente selecciona una acción de forma aleatoria si todas tienen la misma probabilidad, sino se elegirá siempre el de mayor valor a excepción de un porcentaje de veces, equivalente al tanto por uno del valor de épsilon, que se elegirá otra opción aleatoriamente.
Cuando usamos Boltzman si el agente ha propuesto O1…On operadores con sus valores esperados: Q(s, O1),…,Q(s, On) la probabilidad del operador Oi de ser seleccionado es:

$$\frac{e^{Q(s,Oi)/\tau}}{\sum_{j=1}^{n} e^{Q(s,Oj)/\tau}}$$

Donde τ es llamada "temepratura". Temperatura baja lleva a una fuerte influencia en la selección del operador con un valor Q más alto, cuando τ = 0 el mayor valor es siempre seleccionado. Temperaturas altas llevan a una selección más aleatoria, cuando τ = ∞ la selección es totalmente aleatoria. Si τ = 1 el operador es seleccionado en una distribución uniforme en base a los valores Q. Por ejemplo si hay 3 operadores con valores Q de 0.15, 0.1 y 025 la probabilidad de que cada operador sea seleccionado respectivamente es del 30%, 20% y 50%. Dependiendo de los valores de Q, las

recompensas y la temperatura la velocidad de aprendizaje cambia. Sin embargo, excepto cuando τ y Ɛ son 0, las dos maneras tienen cierta probabilidad de seleccionar otros operadores, para asegurar que otras opciones son exploradas y no simplemente ignoradas solo porque un operador parece que tenga mayores opciones de acertar y llevar a la solución rápidamente. SOAR permite modificar estos valores en pos de personalizar el comportamiento.

Actualizar los valores numéricos de las reglas de RL

Cuando llega el momento de actualizar los valores numéricos de las reglas RL exiten 2 procedimientos, SARSA y Q-learning. En ambos procedimientos el valor de la función se actualiza en base a la recompensa recibida por elegir una acción y un "descuento" de la posible futura recompensa esperada.
La actualización utiliza el valor actual y una modificación que es la diferencia entre una futura recompensa esperada y la recompensa recibida actualmente, concretamente es la siguiente:

$$Q(s, O) \leftarrow Q(s, O) + \alpha[r + \gamma Q(s', O') - Q(s, O)]$$

Donde s es el estado actual, s' es el siguiente estado, O es el operador actual y O' es el operador a aplicar en s'. El valor actual Q(s,O) es modificado añadiendo el resto de la ecuación. La recompensa recibida es r y el "descuento" esperado de la futura recompensa es γQ(s',O'). Esto se incluye ya que es incierto cuando o como volverá a ser recibida una recompensa posteriormente. El ratio de aprendizaje α modera el efecto del cambio en la posible recompensa esperada. El descuento y el ratio de aprendizaje tienen valores entre 0 y 1.

Si tenemos la siguiente situación: el agente comienza en el estado S, selecciona el operador O1 en base a las reglas cuyos valores numéricos son: P1=0.34, P2=0.45 y P3=0.02, lo cual combinadas da un Q-valor de 0.81. Tras ser aplicado el operador se recibe una recompensa de 0.54 y los tres operadores siguientes O4, O5 y O6 son propuestos para el siguiente estado S'. Si el operador 6 es seleccionado la actualización (con un valor de 0.63) sucede de la siguiente forma:

Usando SARSA la supuesta recompensa futura se coge para ser el Q-valor del operador siguiente. Tendríamos esto:
$\alpha[0.54 + \gamma * 0.63\text{-}0.81]$. Si usamos γ = 0.9 y α= 0.1 nuestra cuenta da un resultado de : 0.0297.

Usando Q-learning la supuesta recompensa futura es el máximo valor de los operadores propuestos, en este caso tenemos que O4 tenía un valor de 0.75, entonces cuando se usa Q-learning lo haríamos de la siguiente manera:

0.1*[0.54+0.9*0.75-0.81]=0.0405.

Para ambos métodos la actualización es dividida por el número de reglas que han contribuido al Q-valor del operador, en ete caso 3 reglas. Por ejemplo en el caso de SARSA añadiríamos 0.01 a cada regla (0.0297/3). Quedando P1=0.35, P2=0.46 y P3=0.03.
Cuando no es posible realizar estas operaciones porque no se puede conseguir saber un valor futuro esperado, la recompensa es guardada y la actualización se hace cunado es posible suponer la recompensa futura. Esto es lo que ocurre cuando los operadores no tienen preferencias numéricas.

Como una forma sencilla de entender la implementación del RL usaremos un agente sencillo que decide entre moverse en 4 direcciones (Norte, sur, este u oeste) en busca de comida. Se ha creado una "sensación de olor". Cuando el agente decide moverse hacia una u otra dirección se le da una recompensa en función de si se está acercando o no a la comida.

La implementación es la siguiente. Primero vemos las funciones de inicialización del agente:

```
sp "propose*initialize
   (state <s> ^superstate  nil
         -^name)
-->
   (<s> ^operator <o> +)
   (<o> ^name  initialize)
"
sp "apply*initialize
   (state <s> ^operator.name  initialize)
-->
   (<s> ^count 0
      ^mov N
      ^posx  1
      ^posy  1
      ^name cont)
"
```

La propuesta y la aplicación son similares a los ejemplos previos. En la aplicación aprovechamos para crear los elementos que precisaremos para el correcto funcionamiento. Un contador para mantener un seguimiento de la cantidad de decisiones tomadas, la posición X y Y en el mundo bidimensional de nuestro agente y un atributo "^mov" del cual hablaremos en la sección de problemas encontrados y su resolución.

Vamos a ver cómo hemos implementado los operadores de los movimientos del agente:

```
sp "propose*move*N
   (state <s> ^count <c>
         ^name cont
         ^mov N
```

```
        ^io.input-link <il>)
  (<il> ^vacioN si)
-->
  (<s> ^operator <o> + )
  (<o> ^name N)
"
sp "apply*move*N
  (state <s> ^operator <o>
        ^count <c>
        ^posx <x>
        ^posy <y>
        ^mov N
        ^io.input-link <il>
        ^reward-link <r>)
  (<il> ^rewarN <rw>)
  (<o> ^name N)
-->
  (<s> ^mov N - S)
  (<r> ^reward.value <rw>)
  (<s> ^posy <y> - (+ <y> 1))
  (<s> ^count <c> - (+ <c> 1))
"
```

En la regla de proposición en las condiciones nos aseguramos de que estamos en el problema correcto (^name count), utilizamos la variable del contador como control y el input-link, que es necesario para saber si la posición Norte en ese caso está vacía, por lo tanto es elegible, si no lo estuviera sería el caso de estar frente a una pared o un obstáculo. Las acciones serán proponer el operador con una preferencia aceptable y darle un nombre para diferenciar los operadores que puedan ser propuestos en este ciclo. El operador es propuesto con una preferencia aceptable, para que el RL funcione correctamente debemos tener también una preferencia indiferente que haremos utilizando una regla especial posterior. Si no lo hiciéramos y todos los operadores fueran aceptables SOAR entraría en un impasse para intentar elegir el operador más correcto, utilizando otros conocimientos posibles.

Después tenemos la regla de aplicación del operador. Si el algoritmo aleatorio decide que éste debe ser aplicado se ejecuta esta regla. En las precondiciones queremos recuperar del estado el input-link y el reward-link. El input lo necesitamos para sacar el valor de recompensa de haberse movido en esa dirección. Estos valores se calculan dinámicamente en cada ciclo en la aplicación Java principal y se le pasan al agente Soar como si fueran datos recogidos por un sensor. Este dato lo tenemos que añadir a la entidad de recompensas del estado inicial, el reward link, donde se almacena y es usado después para modificar los porcentajes del algoritmo aleatorio de elección de operadores. Lo que tenemos justo después en las acciones, las dos finales es el aumento de 1 unidad en el contador de decisiones y el aumento en el eje Y de una posición, ya que nos movemos al norte. Es importante modificar los valores que correspondan de X e Y ya que estos datos son los que se envían al output, y utiliza nuestra aplicación Java para saber dónde está el agente en todo momento.

La regla que vamos a usar es para añadir el atributo indiferente al operador es especial:

```
gp "rl*move
   (state <s> ^count <c>
         ^name cont
         ^operator <o> +
         ^io.input-link <il>)
   (<o> ^name [N E W S])
   (<il> ^rewarN [1.0 -1.0 1.5 -1.5 2.0 -2.0])
   (<il> ^rewarE [1.0 -1.0 1.5 -1.5 2.0 -2.0])
   (<il> ^rewarW [1.0 -1.0 1.5 -1.5 2.0 -2.0])
   (<il> ^rewarS [1.0 -1.0 1.5 -1.5 2.0 -2.0])
-->
   (<s> ^operator <o> = 1 )
"
```

En orden de cosas a destacar de esta peculiar regla tenemos, en primera instancia el uso de "gp" en vez del típico "sp" como palabra reservada SOAR para indicar el comienzo de una regla. Esto se debe a que va a ser una regla especial en la cual se van a dar unas opciones para los valores de algunas variables y el sistema debe crear de forma automática y combinatoria reglas con todas las posibilidades (todas las combinaciones) dejando las mismas acciones. El valor de ^name, el nombre del operador tiene [N E W S], lo cual indica, debido a los corchetes, que el valor puede ser cualquiera de esos 4, por lo tanto, si solo existiera eso en la regla "gp" se crearían automáticamente 4 reglas, cada una con un valor de ^name, todas con el valor indiferente de la acción. Sin embargo tenemos también todos los valores posibles para las recompensas en N E W S, tanto negativos como positivos. Esto hace que cada regla creada por SOAR se convierta en un posible escenario, por supuesto algunas de ellas son irreales y nunca podrán darse, pero escribir todas las demás a mano es inviable y que haya reglas de más no perjudica al funcionamiento. Cada uno de estos posibles escenarios será recompensado al ser elegido modificando su peso para la próxima vez. De esta manera los primeros intentos el agente se mueve un poco errático y va modificando los pesos de las opciones con que se va encontrando y tras unos pocos intentos al encontrarse en una situación que la recompensa Norte es 2.0 y la Sur -2.0 seguramente tendrá ya los pesos creados para moverse al Norte sin ninguna duda en busca de esa recompensa positiva. De esta manera el agente a través del sistema es capaz de emular un comportamiento de aprendizaje.

Por último el valor indiferente es inicializado a 1 por elección propia, este valor no tiene por qué ser necesariamente ese, pero para lidiar con las recompensas 1, 1.5 y 2 se comporta de forma adecuada y por eso ha sido utilizado.

Los otros 3 conjuntos de reglas para moverse al este, sur y oeste son similares.

Vamos a ver las reglas de llegada a meta:

```
sp "goal*reached*propose
   (state <s> ^count <c>
          ^mov N
          ^name cont
          ^io.input-link <il>)
   (<il>  ^comidaX <cx>
         ^comidaY <cy>)
   (<s>  ^posx <cx>
         ^posy <cy>)
-->
   (<s> ^operator <o> + )
   (<o> ^name Fin)
"

sp "better*goal
   (state <s> ^count <c>
          ^name cont
          ^operator <o> + )
   (<o> ^name Fin)
-->
   (<s> ^operator <o> > )
"
```

En esta regla en las precondiciones tenemos que encontrar la forma de saber que hemos alcanzado el objetivo. En este caso este agente alcanzará su objetivo en el momento en que alcance la comida, situada en un punto aleatorio por la aplicación y cuyos valores de X e Y le hemos pasado al agente por el input link con los nombres de atributo de "comidaX y comidaY". Por lo tanto los leemos y guardamos su valor y obligamos a que "posX y posY" que es la posición actual del agente sean iguales a ellas. Si es así el agente ha llegado a su destino. Se propone un operador nuevo Fin con atributo aceptable para que maneje el final del agente, sin embargo solo con eso nos encontraríamos con que hubiera una probabilidad de no ser elegido, por lo tanto vamos a utilizar el atributo "better" (>) además del aceptable para asegurarnos de que se elige este operador sobre los demás.

```
sp "goal*reached*apply
   (state <s> ^operator <o>
          ^count <c>
          ^mov N
          ^reward-link  <r>)
   (<o> ^name Fin)
-->
   (<s> ^mov N - F)
   (<r> ^reward.value  10)
   (<s> ^count <c> - (+ <c> 1))
"
```

La aplicación del operador Fin modifica el estado general del problema (manejado por la variable ^mov) a un estado especial y da una recompensa mucho mayor por conseguir la meta.

Como podemos ver no está aquí la regla que contiene la acción "halt" necesaria para finalizar un agente, ya que existe un problema con las recompensas del que se habla en el Anexo: "problemas encontrados y su solución" y donde además se encuentran 4 reglas adicionales de este agente, necesarias para su buen funcionamiento.

Los nuevos valores de la ecuación de aleatoriedad perduran tras el fin del agente y si es iniciado de nuevo desde ese punto, como haremos, tendrá los valores anteriores.

Las últimas reglas del agente son las de monitorización de la actividad y la gestión del output.

```
sp "monitor*posy
   (state <s> ^posy <y>)
-->
   (write (crlf) |posy = |<y>)
"
sp "monitor*posx
   (state <s> ^posx <x>)
-->
   (write (crlf) |posx = |<x>)
"
sp "output*count
   (state <s> ^io.output-link <ol>
         ^count <c>
         ^posx <x>
         ^posy <y>)
-->
   (<ol> ^pos.posx <x> ^pos.posy <y>)
"
```

La monitorización simplemente escribe cada vez que hay un cambio en posX o posY, como control del funcionamiento del agente. Y la regla de output utiliza los valores X e Y y el output-link para pasarle a éste último los valores, de la manera concreta en la que funcionará con Java en la aplicación para que puedan ser leídos los valores. De esta manera "posx y posy" son los nombres que tiene también la clase de java de recuperación de éstos, con lo que el "handler" puede hacer las correspondencias.

Agente con Reinforcement Learning en Java + SOAR

Las librerías en Java proporcionan las herramientas necesarias para crear un agente, cargar las reglas necesarias desde un archivo de texto y gestionar los inputs y outputs de SOAR para interaccionar como si la aplicación simulase el mundo exterior y sus sensores o actuadores.

Existen dos formas de crear un agente externo, un agente que se ejecuta de forma secuencial en el programa Java y un agente que se ejecuta en su propio thread al margen. Ambas maneras deben accederse de la misma forma, mediante "Listeners" programados como interrupciones que mandará SOAR para que el agente sepa cuando

está en momento de ciclo: output-input y ejecutar el código escrito previamente para ello.

Para este agente se ha decidido el uso de un agente que tenga su propio "thread". En los "Listeners" gestionaremos los datos que podremos en el input-link y los que recogemos del output-link. Los datos a insertar en input-link deben tener un tipo específico para que el sistema comprenda lo que tiene que hacer, por eso usamos tipos que nos dan las librerías, en este caso concreto sobre todo el tipo "StringElement" "IntElement" o "FloatElement". La primera vez que tenemos que crear las entidades con sus atributos en input-link las añadimos utilizando el método del agente Create[tipo]WME, pasándole el identificador del inputlink. Al añadirlas nos devuelve el procedimiento un "puntero" a nuestros recién creados valores que posteriormente podremos modificar solo cuando nos encontremos en la interrupción de input.

La gestión del input y el output, como decimos, solo es posible cuando el ciclo de SOAR se encuentra parado tras la fase de output y previamente al comienzo de un nuevo ciclo, que empieza con el input. Por ello necesitamos frenar cada ciclo para realizar las modificaciones pertinentes en el estado del problema, recoger los valores del output y actualizar el input. Para ello vamos a utilizar un "Event Listener", Java reaccionará ante una interrupción de SOAR cuando éste alcance el punto deseado, lo parará y permitirá ejecutar el código Java dentro del "handler" (gestor) de la interrupción. Dentro de esta se gestionará el output, el "repaint" de la interfaz y el input del problema. Tras esto el agente vuelve a la normalidad.

Código del agente (Java)

Vamos a ver la clase principal, donde se ve la utilización de las librerías SML en comunión con Java para gestionar Soar mientras se gestiona una interfaz gráfica de usuario. Las funciones con un comentario "…" y los valores "X" han sido añadidos en sustitución de valores y códigos que no se considera relevante mostrar aquí.

```
//cabeceras
package …;
import java.awt.*;
import javax.swing.JPanel;
import sml.*;


public class Execu extends JPanel {


  //Identificadores de los elementos de Input.
  private StringElement nvc, svc, evc , wvc;
  private IntElement comX, comY;
  private FloatElement nvl, svl, evl, wvl;
   //Identificadores para la posición del agente y su valor de dibujo.
```

```
    private int x, y, dx, dy;
    //Clases tablero y ventana.
    private Tablero ent;
    private Ventana ventana;
    // Kernel y el Agente.
    private Kernel kernel;
    private Agent agent;
    //Getters de los valores de posicion del agente.
    public int getX (){
            return this.x;
    }
    public int getY (){
            return this.y;
    }

    //**************CONSTRUCTOR***************
    //Cosntructor de la clase, creción del agente SOAR
            public Execu(Ventana laVentana) {
               // Constructor
               ventana = laVentana;
               kernel = Kernel.CreateKernelInNewThread(-1);
                        if (kernel.HadError()) {
                           //Problema en la creacion del kernel
                           System.out.println("Error creating kernel: " + kernel.GetLastErrorDescription()) ;
                           System.exit(1);
                        }
                        agent = kernel.CreateAgent("RL");
            }

//**************INICIALIZADOR*************
            public void start() throws InterruptedException {
            //Inicializa los valores, inicializa la GUI y llama a iniciar Soar
                        ent = new Tablero();
                        ent.rellenartablero();
                        drawTapete(getGraphics());
                        drawComida(getGraphics());
                        x = X; y = X; convertirAGraficos();
                        drawAgente(getGraphics());
                        agent = this.SetupSoar();
            }

            public void reStart() {
            //Reinicializa el agente para la siguiente ejecución
                        x=X;y=X;
                        drawTapete(getGraphics());
                        drawComida(getGraphics());
                        convertirAGraficos();
                        drawAgente(getGraphics());
                        System.out.println(agent.ExecuteCommandLine("init-soar"));
            }

//*************FUNCIONES DE DIBUJO*************
            public void drawTapete(Graphics graphics) {
                        //...
            }
            public void drawComida(Graphics graphics) {
                        //...
```

```
        }
        public void convertirAGraficos(){
                //...
        }
        public void drawAgente(Graphics graphics) {
                //...
        }


//*********CREAR EVENTOS Y CARGAR PROGRAMA SOAR*********
        public Agent SetupSoar() throws InterruptedException {

           //linicializar -> LISTENERS para mostrar traza (print) y gestión de IO (update)
           EventListener listener = new EventListener();
        agent.RegisterForPrintEvent(smlPrintEventId.smlEVENT_PRINT, listener, null);
        kernel.RegisterForUpdateEvent(smlUpdateEventId.smlEVENT_AFTER_ALL_OUTPUT_PHASES,
        listener, null) ;

        // Cargar el programa Soar (archivo de reglas)
        boolean load = agent.LoadProductions("aprendizajeRL.soar");
         if (!load || agent.HadError()) {
                throw new IllegalStateException("Error loading productions: " +
                agent.GetLastErrorDescription());
         }

        //Si ponemos autocommit a false cuando actualicemos los valores del input se mandaran como
        //un único pack cuando hagamos commit, en vez de cuando elija el agente, evitando errores.
         kernel.SetAutoCommit(false);
        //Mandamos comandos con ExecuteCommandLine, en este caso aprendizaje RL encendido
         System.out.println(agent.ExecuteCommandLine("rl --set learning on"));

        //Vamos a poner los valores en el input-link iniciales
                 Identifier pInputLink = agent.GetInputLink() ;

                comX = agent.CreateIntWME(pInputLink, "comidaX", X) ;
                comY = agent.CreateIntWME(pInputLink, "comidaY", X) ;
                nvc = agent.CreateStringWME(pInputLink, "vacioN", X);
                nvl = agent.CreateFloatWME(pInputLink, "rewarN", X);
                svc = agent.CreateStringWME(pInputLink, "vacioS", X);
                svl = agent.CreateFloatWME(pInputLink, "rewarS", X);
                evc = agent.CreateStringWME(pInputLink, "vacioE", X);
                evl = agent.CreateFloatWME(pInputLink, "rewarE", X);
                wvc = agent.CreateStringWME(pInputLink, "vacioW", X);
                wvl = agent.CreateFloatWME(pInputLink, "rewarW", X);
                agent.Commit();

        //Ejecutamos el agente
                agent.RunSelfForever();
                return agent;
        }


//***************LISTENERS Y HANDLERS**********************
public static class EventListener implements Agent.PrintEventInterface,Kernel.SystemEventInterface{

                public void printEventHandler(int eventID, Object data, Agent agent, String message) {
                        System.out.println("Received print event in Java: " + message) ;
```

```
		}
		public void updateEventHandler(int eventID, Object data, Kernel kernel, int runFlags) {
		//Aquí se va a gestionar el output, repintado de GUI e input.
			recogerOutput();
			ventana.updatePuntuacion();
			drawTapete(getGraphics());
			drawComida(getGraphics());
			convertirAGraficos();
			drawAgente(getGraphics());
			modificarInput();
		}

}

	//Finalizar el agente Soar y salir de la aplicación Java
	public void salir(){
		System.out.println(agent.ExecuteCommandLine("excise --all"));
		kernel.Shutdown();
		kernel.delete();
		System.exit(0);
	}

	public void modificarInput(){
		//Gestión de Input
		Celda[] vector=ent.calcularAdyacentes(x, y);
		vector = ent.calcularRewards(x, y, vector);
		if (vector[0].isVacia()){
			agent.Update(nvc, X);
			agent.Update(nvl, vector[0].getValor());
		} else {
			agent.Update(nvc, X);
		}
		//...

		agent.Commit();
	}

	public void recogerOutput(){
		//Gestión de Output
		for (int i = 0; i < agent.GetNumberCommands(); ++i) {
			Identifier commandWME = agent.GetCommand(i);
			String commandName = commandWME.GetAttribute();
			if (commandName.equals("pos")) {
				String posicion = commandWME.GetParameterValue("posx");
				if (posicion!=null){
					x=Integer.parseInt(posicion);
				}
				posicion = commandWME.GetParameterValue("posy");
				if (posicion!=null){
					y=Integer.parseInt(posicion);
				}}commandWME.AddStatusComplete();}}
```

Anexo D

Sistema de memoria: Semantic Memory

La Semantic Memory es la memoria de la arquitectura que sería el equivalente a las memorias grandes y lentas, como un disco duro, en contraposición con la memoria RAM a la cuál equivaldría la Working memory. Esto significa que su papel es guardar toda la información que un agente pueda necesitar en un futuro, pero que no necesita tener siempre a mano, y permitir la descarga de esta información a la Working memory cuando sea preciso para mejorar el conocimiento sobre algo. Se puede expresar que esta memoria guarda los conceptos generales que un agente tiene o ha aprendido, ya que esta memoria tiene 2 formas de funcionar: con memoria precargada y aprendiendo automáticamente.

Por ejemplo, podríamos tener la situación de que necesitáramos saber cuándo sucedió algo, por ejemplo tienes una foto de Central Park justo al anochecer de ese mismo día, y quieres saber a qué hora se hizo la foto. Pensaríamos en el Central Park, por lo que sabemos seriamos capaces de llegar a la conclusión de que está en Nueva York, en EEUU, cuyo huso horario tiene una diferencia D con el nuestro, el atardecer es a una hora X que correspondería a la hora Y de nuestro país, por lo tanto podemos saber la hora. Para llegar a esta conclusión hemos utilizado muchos datos que tenemos memorizados. Podríamos decir que "accedemos" a ellos para llegar a la conclusión precisa. El sistema de Semantic Memory funciona de manera muy similar a este ejemplo.

La forma de almacenar estos datos es similar a la Working memory. Son almacenados los "arboles" de entidades con atributos, sin guardar información adicional sobre cuando son almacenados o en qué orden, ya que, al contrario que con la Episodic Memory, esto no es relevante.

Para recuperar la información se crea una consulta basada en un "árbol" o "subárbol" del estado de la Working memory actual, si se encuentra un conjunto de entidades con sus atributos que se enlaza perfectamente con la consulta entonces se descarga directamente la información desde la memoria a largo plazo a la de corto plazo, añadiendo todo lo que no hubiera previamente en ésta última.

Vamos a mostrar un ejemplo básico visual de su funcionamiento empezando por información guardada en la Semantic Memory (ver Figura a4.1 y a4.2) .

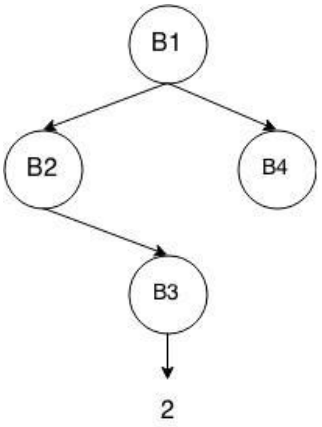


Figura a4.1 *– Diagrama de concepto 1*

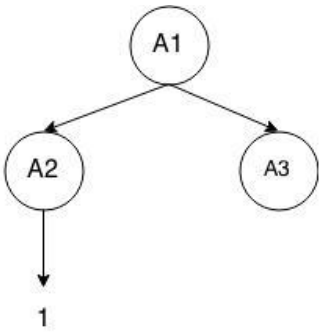


Figura a4.2 *– Diagrama de concepto 2*

Vemos 2 estructuras de dos conceptos constituidos por entidades con sus atributos (el nombre de los atributos no aparece para simplificar). Ambas son independientes, son conocimientos sobre algo que guardamos. En un momento dado nos encontramos con la siguiente situación de Working memory (ver Figura a4.3)

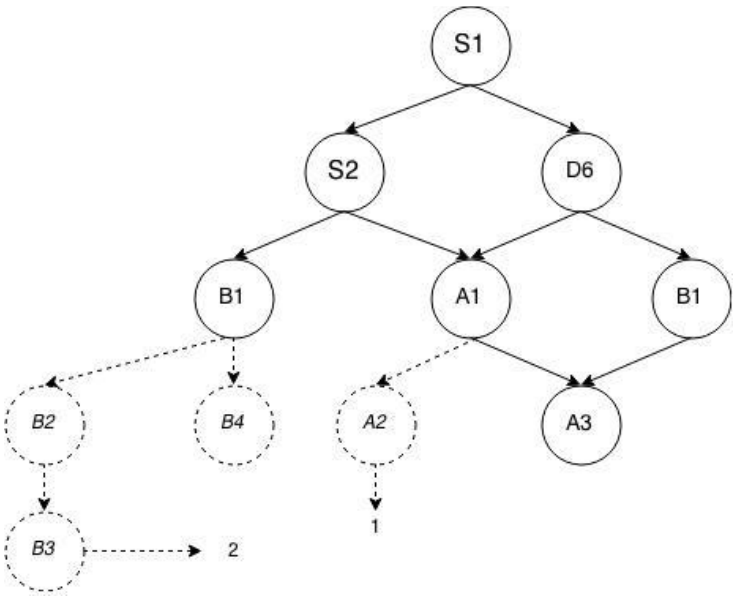


Figura a4.3 *– Diagrama de WM en el estado actual*

El grafo con líneas continuas es la situación de la Working memory. Se ha añadido al grafo con líneas discontinuas cómo se adecuaría la información que teníamos almacenada en la Semantic memory si el agente decidiera recuperar esa información. Añadiendo todas esas entidades que faltaban al original, creando un mayor conocimiento.

De esta manera es muy posible que ciertas reglas que antes no pudieran ser propuestas ya que no correspondían a la situación actual sean posibles opciones, ya que ahora se contará con una información mucho más extensa.

En la práctica nos encontramos lo siguiente: desde la creación de un nuevo estado en la Working memory la arquitectura crea las siguientes entidades para facilitar la interacción del agente con la Semantic Memory:

(<s> ^smem <smem>)
(<smem> ^command <smem-c>)
(<smem> ^result <smem-r>)

Ciertas reglas utilizan la estructura de "command" para acceder o cambiar la memoria, mientras que las consultas devueltas se guardaran en la entidad de "result". Las acciones de las reglas no deben eliminar la estructura del resultado directamente, ya que está ligada a la Semantic memory. Los agentes guardan identificadores de "largo plazo" en la Semantic Memory cuando crean un comando de almacenaje, esto es una entidad que tiene un enlace con una estructura de Semantic Memory, su atributo es "store" y su valore es otro identificador. *<s> ^smem.command.store <identifier>.*

Un agente recupera información de ésta memoria creando un comando (una consulta) apropiada en la entidad "smem" de la estructura del estado principal (incial). Al final de la fase de output de cada decisión la Semantic memory procesa las estructuras de comando. Se añaden los resultados, meta-datos y errores a la estructura de "result" en el "smem". Solo puede ser recuperada una estructura por ciclo y las consultas mal escritas o mal formadas conllevaran a la creación de la entidad: *<s> ^smem.result.bad-cmd <smem-c>.* Tras el procesamiento de la consulta la Semantic Memory la ignorará hasta el momento en que algo de la consulta cambie, cuando esto ocurra el resultado anterior se borra y se procesa la nueva consulta.

El sistema de búsqueda intenta encajar la consulta creada con cualquier conjunto de entidades guardadas hasta que una encaja perfectamente, aun teniendo entidades adicionales a la estructura buscada. Una consulta está compuesta de entidades que describen los "objetos" y "atributos" de un identificador a largo plazo, explicados anteriormente. Si contiene un valor constante debe encontrarse una similitud perfecta del valor, igual que con un identificador de largo plazo. Si el identificador es normal solo se tendrá en cuenta el nombre del atributo, siendo su valor no relevante, ya sea un valor constante u otro identificador. La consulta se escribe en: *<s> ^smem.command.query <cue>.*

Veamos una regla donde se crea una consulta:

```
sp {smem*sample*query
(state <s> ^smem.command  <sc>
           ^lti <lti>
           ^input-link.foo <bar>)
-->
(<sc>     ^query <q>)
(<q>      ^name <any-name>
```

^foo <bar>
^associate <lti>
^age 25)}

En este ejemplo el identificador <lti> tendrá que corresponder con un identificador a largo plazo y <bar> corresponderá con una constante. Así que la consulta deberá devolver un identificador a largo plazo cuyas entidades satisfagan una correspondencia con los siguientes requisitos: un atributo "name" con un valor cualquiera, un atributo "foo" con un valor igual al valor de la variable <bar> en el momento en que se recupere de la memoria, un atributo "associate" con un valor igual al del identificador de largo plazo en el momento y un atributo "age" con un valor entero de 25.

Si no hay un solo identificador a largo plazo que cumpla todos estos requisitos se devolverá un error de la forma: *<s> ^smem.result.failure <cue>.* Y sino se devolverán los arboles de entidades siguientes:
<s> ^smem.result.success <cue>
<s> ^smem.result.retrieved <retrieved-lti>

Ejemplo de uso de Semantic Memory

Lo primero vamos a ver como cargar conocimiento en la Semantic memory y ver su contenido. Para esto como ya hemos explicado anteriormente la estructura:

```
smem --add {
(<a> ^name alice ^friend <b>)
(<b> ^name bob ^friend <a>)
(<c> ^name charley)
}
```

Cuando ejecutamos este comando 3 objetos se añaden a la Semantic memory. Podemos ver el contenido usando el comando: *smem –print*. El cuál tras ejecutar el comando anterior mostrará lo siguiente:

```
(@A1 ^friend @B1 ^name alice [+1.000])
(@B1 ^friend @A1 ^name bob [+2.000])
(@C3 ^name charley [+3.000])
```

Las variables del comando –add deben ser inicializadas como identificadores específicos y precedidos por @, ya que serán identificadores a largo plazo (LTIs). Todo el conocimiento de la Semantic memory no está conectado directamente o indirectamente con ningún estado.

Ahora que hemos visto el contenido de la memoria podemos percibir que nada de este conocimiento está presente en ninguna de las otras memorias del agente. Y además es modificable, si quisiéramos limpiar la memoria de golpe usamos el comando: *smem –init.*

Ya hemos hablado previamente de la estructura fija que tiene el estado inicial para soportar la información recuperada de ésta memoria, pero como todos los mecanismos de SOAR éste debe ser iniciado al principio con el comando: *smem –set learning on*.

Para guardar información en la Semantic Memory directamente desde el agente utilizamos el comando "store". La forma de utilizarlo es: *<cmd> ^store <id>)* donde <cmd> es el enlace "command" de un estado y <id> es un identificador. Un agente puede ejecutar varios comandos de almacenaje simultáneamente, estos son procesados al final de la fase donde se han escrito. Si el identificador no es a largo plazo se cambia a uno de estos, y si ya existe en la memoria todas las entidades son actualizadas (sobrescritas). Ejemplo:

```
sp {propose*init
(state <s> ^superstate nil
           -^name)
-->
(<s> ^operator <op> +)
(<op> ^name init)}
sp {apply*init
(state <s> ^operator.name init
           ^smem.command <cmd>)
-->
(<s> ^name friends)
(<cmd> ^store <a> <b> <c>)
(<a> ^name alice ^friend <b>)
(<b> ^name bob ^friend <a>)
(<c> ^name charley)}
sp {propose*mod
(state <s> ^name friends
           ^smem.command <cmd>)
(<cmd> ^store <a> <b> <c>)
(<a> ^name alice)
(<b> ^name bob)
(<c> ^name charley)
-->
(<s> ^operator <op> +)
(<op> ^name mod)}
sp {apply*mod
(state <s> ^operator.name mod
           ^smem.command <cmd>)
(<cmd> ^store <a> <b> <c>)
(<a> ^name alice)
(<b> ^name bob)
(<c> ^name charley)
-->
(<a> ^name alice -)
(<a> ^name anna
     ^friend <c>)
(<cmd> ^store <b> -)
(<cmd> ^store <c> -)}
```

Al ejecutar estas reglas mostrando la información de la Semantic Memory podremos ver la siguiente traza (en el debugger):

--- apply phase ---
--- Firing Productions (PE) For State At Depth 1 ---
*Firing apply*init*
--- Change Working Memory (PE) ---
=>WM: (25: C3 ^name charley)
=>WM: (24: B1 ^friend A1)
=>WM: (23: B1 ^name bob)
=>WM: (22: A1 ^friend B1)
=>WM: (21: A1 ^name alice)
=>WM: (20: C2 ^store A1)
=>WM: (19: C2 ^store B1)
=>WM: (18: C2 ^store C3)
=>WM: (17: S1 ^name friends)
--- Change Working Memory (PE) ---
=>WM: (28: R3 ^success @A1)
=>WM: (27: R3 ^success @B1)
=>WM: (26: R3 ^success @C3)

Al ejecutar "apply*init" se han añadido los tres comandos de almacenaje a la Working memory, los identificadores en un principio no a largo plazo son similares a los del ejemplo inicial. Entonces al final de la fase de elaboración se procesan los comandos, convirtiendo los identificadores en largo plazo, añadiendo el estatus a cada comando. El estado final será similar al mostrado en el ejemplo inicial en la memoria.

La aplicación del siguiente operador modifica el contenido de la memoria sobrescribiendo el contenido de un identificador de largo plazo (@A1).

*Firing apply*mod*

--- Change Working Memory (PE) ---
=>WM: (33: @A1 ^name anna)
=>WM: (32: @A1 ^friend @C3)
<=WM: (21: @A1 ^name alice)
<=WM: (18: C2 ^store @C3)
<=WM: (19: C2 ^store @B1)
--- Change Working Memory (PE) ---
<=WM: (26: R3 ^success @C3)
<=WM: (27: R3 ^success @B1)

Los comandos "store" de almacenaje para @B1 y @C3 son eliminados por la aplicación del operador mod, y la entidad de @A1 también es eliminada y añadida otra. Al final la memoria limpia toda la información de los, ya viejos, comandos "store". Ahora al mostrar la memoria:

(@A1 ^friend @B1 @C3 ^name anna [+4.000])
(@B1 ^friend @A1 ^name bob [+2.000])
(@C3 ^name charley [+3.000])

Vemos que la entidad de @A1 ya ha sido modificada, sin embargo @B1 y @C3 todavía no.

Ahora vamos a tratar la recuperación de la información en base a consultas. El agente pide a la memoria todas las entidades de un identificador de largo plazo desconocido, descrito por un sub-arbol de estas entidades. Como hemos expuesto anteriormente usamos el comando: *(<cmd> ^query <cue>)*, donde las entidades tendrán <cue> como su identificador. Tenemos estas 2 reglas:

```
sp {propose*cb-retrieval
(state <s> ^name friends
           ^smem.command  <cmd>)
(<cmd> ^retrieve)
-->
(<s> ^operator <op> + =)
(<op> ^name cb-retrieval)}
sp {apply*cb-retrieval
(state <s> ^operator <op>
           ^smem.command  <cmd>)
(<op> ^name cb-retrieval)
(<cmd> ^retrieve  <lti>)
-->
(<cmd> ^retrieve  <lti> -
          ^query <cue>)
(<cue> ^name <any-name>
          ^friend <lti>)}
```

Estas reglas recuperan un identificador que tiene 2 condiciones: 1 tiene una entidad cuyo atributo es “name” pero el valor puede ser cualquiera y tiene una entidad cuyo atributo es “friend” y el valor es de largo plazo recuperado como resultado de aplicar las reglas comentadas anteriormente. Recordemos que solo se puede hacer la recuperación de un comando por cada ciclo.

El resultado tras aplicar estas reglas (en el “smem” del estado inicial) es:

```
(S2 ^command C2 ^result R3)
(C2 ^query C4)
(C4 ^friend @C3 ^name A2)
(@C3 ^name charley)
(R3 ^retrieved @A1 ^success C4)
(@A1 ^friend @B1 ^friend @C3 ^name anna)
```

Vemos que de la memoria se ha recuperado y añadidod a la working memory el identificador @A1 con todas sus entidades, si hubiésemos recuperado @B1 hubiésemos visto lo siguiente:

```
(S2 ^command C2 ^result R3)
(C2 ^query C4)
(C4 ^friend @B1 ^name A2)
(@B1 ^friend @A1 ^name bob)
(R3 ^retrieved @A1 ^success C4)
```

(@A1 ^friend @B1 ^friend @C3 ^name anna)

Como ningún identificador a largo plazo satisface la consulta el estatus será “failure” (fallido) y no se devolverá ninguna estructura. Si múltiples identificadores hubiesen satisfecho las condiciones se hubiera devuelto el conjunto de entidades con una valoración más alta. Esta valoración no es más que un entero que se va incrementando para mostrar si se ha almacenado o recuperado el objeto más o menos recientemente. Es posible prohibir la recuperación de algunos identificadores en concreto.

Anexo E

Sistema de memoria: Episodic Memory

La Episodic Memory es la encargada de guardar una "serie" de experiencias. Es lo que nosotros "recordamos" en contraposición con la Semantic Memory, que es lo que "sabemos". Consiste en cosas concretas libres del contexto específico en el que fueron "aprendidas" y son útiles en el razonamiento sobre las propiedades generales del mundo del agente. Hace posible extraer información y procesos recurrentes que quizá no habíamos visto en la experiencia origen y combinarlo con el conocimiento de ese momento. La Episodic Memory tiene una estructura temporal que hace posible recuperar una secuencia de episodios que pueden ser utilizados para predecir comportamiento y dinámicas del medio en situaciones similares.

La Episodic memory es algo que los humanos, excepto cuando un accidente o una enfermedad lo impiden, damos por hecho. Provee una memoria de eventos previos y mantiene otras capacidades cognitivas adicionales que hacen posible el razonamiento y aprendizaje de un agente inteligente.

El rol de esta memoria en el problema es el mismo que la Semantic memory, provee de una memoria adicional de largo plazo que puede ser accedida para enriquecer el estado actual. El uso estándar es acceder en la aplicación de operadores, aunque puede ser accedida en cualquier momento. Uno de los aspectos importantes es que mientras usamos Chunking puede ser recogida la información en un sub-estado por lo que en futuros momentos similares no será necesario volver a acceder ya que estará codificado en la regla creada por el "chunk".

Sobre la Episodic memory en SOAR hay que decir que los "episodios" son guardados automáticamente. Cada episodio es una captura (foto) de la Working memory (de sus elementos) que existen en el momento que es tomada. Los episodios incluyen información temporal, por lo que cuando un episodio ha sido recogido se puede acceder a los anteriores.

Para recoger un "episodio" SOAR prepara una "indicación" en el estado de la working memory. Durante la recuperación de la información la captura que mejor se adapte a esa "indicación" es encontrada y reconstruida en un link del estado original donde también se encontraba la inficación. A partir de este momento las reglas del agente pueden utilizar toda la nueva información construida para ejecutar sus acciones o proponer operadores.

Para que funcione la Episodic Memory al usarla en SOAR debemos habilitarla nada más iniciar el agente, ya que por defecto todas las técnicas de aprendizaje y memoria están deshabilitadas, usamos el comando: `epmem --set learning on.`

Por defecto la Episodic memory guarda nuevos episodios en cualquier momento en el que una nueva entidad WME es añadida a la Working memory y tenga como identificador: output-link.
Sin embargo SOAR soporta el guardado de episodios en cada ciclo de decisión ("dc"), el cual se habilita usando el siguiente comando: `epmem --set trigger dc`.

El siguiente parámetro importante es la fase durante la cual se guardan los episodios y los procesos de recuperación. Por defecto este proceso ocurre al final de la fase de output. Pero esta soportado que sea al final de la fase de decisión, usando el comando: `epmem --set phase selection`.

En la representación de la Episodic Memory en el estado principal encontramos diferentes entidades, vamos a ver una representación gráfica de su organización (ver Figura a5.1)

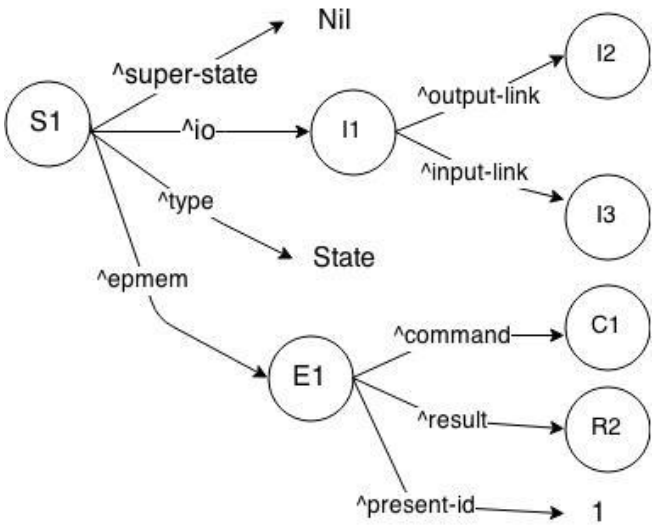


Figura a5.1 – Diagrama de estado inicial con detalle de la EM

El método básico por el que un agente puede recuperar conocimiento de esta memoria se llama recuperación "cue-based". El agente le pide a la memoria un episodio que sea el que más se parezca a un grupo de entidades con atributos presentes en la Working memory. Ese grupo tendrá que estar "colgado" del ^command de la ^epmem, en nuestro ejemplo gráfico, colgado de C1.
Conceptualmente, la memoria compara ese grupo con todos los episodios guardados, dándoles una puntuación y devuelve el episodio más reciente con la máxima puntuación.

EL valor de "present-id" es un entero y se actualiza para mostrar al agente el número de episodio actual. La Episodic memory solo soporta entidades cuyos atributos son constantes, el comportamiento esta indefinido cuando intentamos guardar una entidad cuyo atributo es un identificador. Se utiliza "SQLite" para facilitar la estandarización y eficiencia del guardado y consulta de episodios. Si un comando de

consulta está mal formado resultará en un error que se verá aquí: *<s> ^epmem.result.status bad-cmd.*

Los operadores pueden ser copiados de la memoria, pero no podrán ser utilizados para enlazar con ninguna regla, ya que se les modifica el nombre de atributo de "^operator" a "^operator*" para que esto no pueda ser posible. Todas las consultas deben contener una consulta alojada en: *<s> ^epmem. Command.query <required-cue>*, sin embargo opcionalmente también pueden tener una consulta negada, que hace que los episodios que se emparejen con ella "pierdan puntos" en comparación con los que sí que vayan bien con la consulta normal. La consulta negada se alojará en: *<s> ^epmem.command.query <optional-negative-cue>.*

El proceso de recuperación puede ser comparado conceptualmente con la búsqueda del vecino más cercano. Primero todos los episodios candidatos, definidos como los episodios que como mínimo tienen una hoja que se puede identificar con la consulta. Dos valores son calculados para cada episodio candidato con respecto a la consulta recibida: la "cardinalidad" del emparejamiento (definido como el número de hojas "emparejadas") y la activación del emparejamiento (definido como la suma de los valores de activación de cada hoja emparejada). Cuando se aplica a la consulta negativa todos estos valores van en negativo. Para sacar el valor de emparejamiento de cada episodio estos valores se combinan con respecto a los valores de balance como sigue:
(balance)(cardinality)+(1-balance)*(activation)*

Ejemplo de uso de Episodic Memory

Vamos a hacer un conjunto de reglas SOAR simple que muestre el funcionamiento de la Episodic Memory. Crearemos una "situación" inicial que se guardara automáticamente en el episodio 1 ya que SOAR tendrá habilitada la opción de uso de "EpMem". Reglas necesarias:

```
sp {propose*init
  (state <s> ^superstate  nil
        -^name)
-->
  (<s> ^operator <op> + =)
  (<op> ^name init)
}
sp {apply*init
  (state <s> ^operator <op>)
  (<op> ^name init)
-->
  (<s> ^name epmem
     ^feature2  value
     ^feature  value3
```

```
      ^id <e2>
      ^id <e3>
      ^other-id <e4>)
  (<e2> ^sub-feature value2)
  (<e3> ^sub-id <e5>)
  (<e4> ^sub-id <e6>
       ^sub-feature value2)
}
```

Inmediatamente después propondremos un operador que intente recuperar un episodio con una estructura parecida a la que teníamos como situación inicial. Reglas necesarias:

```
sp {epmem*propose*cbr
  (state <s> ^name epmem
         -^epmem.command.<cmd>)
-->
  (<s> ^operator <op> + =)
  (<op> ^name cbr)
}
sp {epmem*apply*cbr-query
  (state <s> ^operator <op>
         ^epmem.command  <cmd>)
  (<op> ^name cbr)
-->
  (<cmd> ^query <n1>)
  (<n1> ^feature value
      ^id <n2>)
  (<n2> ^sub-feature value2
       ^sub-id <n3>)
}
```

Al ser el único operador propuesto será seleccionado y creará la consulta. Después SOAR se encargará de buscar en la memoria y llegará a la conclusión de que su episodio 1 es suficientemente parecido a la consulta, por lo que nos lo devolverá en el atributo pertinente de nuestra Working memory. Parte de la traza del debugger al ejecutar estas reglas muestran como queda finalmente:

```
step
    1: O: O1 (init)
--- apply phase ---
--- Firing Productions (PE) For State At Depth 1 ---
Firing apply*init
--- Firing Productions (IE) For State At Depth 1 ---
Firing epmem*propose*cbr
Retracting propose*init
--- output phase ---
Initializing episodic memory database in cpu memory.
Erasing contents of episodic memory database. (append = off)
New episodic memory recorded for time 1.
--- input phase ---
=>WM: (33: E1 ^present-id 2)
<=WM: (7: E1 ^present-id 1)
--- propose phase ---
--- decision phase ---
=>WM: (34: S1 ^operator O2)
    2: O: O2 (cbr)
```

```
step
--- apply phase ---
--- Firing Productions (PE) For State At Depth 1 ---
Firing epmem*apply*cbr-query

print --depth 5 e1
(E1 ^command C1 ^present-id 3 ^result R2)
 (C1 ^query N1)
  (N1 ^feature value ^id N2)
   (N2 ^sub-feature value2 ^sub-id N3)
 (R2 ^cue-size 3 ^graph-match 0 ^match-cardinality 2 ^match-score 2,
     ^memory-id 1 ^normalized-match-score 0,6666666666666666 ^present-id 3
     ^retrieved R4 ^success N1)
  (R4 ^feature value3 ^feature2 value ^id I5 ^id I6 ^io I4 ^name epmem
      ^operator* O5 ^other-id O4 ^reward-link R5 ^superstate nil ^svs S5
      ^type state)
   (I5 ^sub-feature value2)
   (I6 ^sub-id S6)
   (I4 ^input-link I7 ^output-link O6)
   (O5 ^name cbr)
   (O4 ^sub-feature value2 ^sub-id S7)
   (S5 ^command C4 ^spatial-scene S8)
    (S8 ^id world)
```

Al final de la traza se ha hecho un "Print" (muestra por pantalla) de E1 que es la rama de la cual cuelgan los resultados de la consulta a la Episodic Memory en nuestra Working memory, vemos resaltado en verde como el "present-id" nos muestra que estamos en el momento 3 , que quedará representado con este número en la Episodic Memory, sin embargo el "memory-id" es 1, que es el número de la "foto" recuperada de la memoria, en este caso nada más inicializar. En la traza se ve el momento en el que el que es guardado ese episodio.

Hay una opción que nos da SOAR para versatilidad de este sistema que consiste en, teniendo recuperado este episodio podemos en el siguiente ciclo recuperar el siguiente guardado a él o el anterior, esto nos puede servir para recuperar información sobre qué ha sucedido después, al tomar cierta decisión, o que decisión ha llevado al estado recuperado, según se recupere el episodio previo o posterior. Las Reglas necesarias serían (en el caso de recuperar posterior):

```
sp {epmem*propose*next
  (state <s> ^name epmem
         ^epmem.command.query)
-->
  (<s> ^operator <op> + =)
  (<op> ^name next)
}
sp {epmem*apply*next
  (state <s> ^operator <op>
         ^epmem.command <cmd>)
  (<op> ^name next)
  (<cmd> ^query <q>)
```

```
-->
  (<cmd> ^query <q> -
       ^next <next>)
}
```

Nuestra trama inmediatamente posterior a la mostrada anteriormente usando estas reglas adicionales pedirían recuperar ese episodio siguiente y al hacer "Print" de nuevo veríamos el episodio 2 en el "memory-id":

```
Firing epmem*propose*next
Retracting epmem*propose*cbr
--- output phase ---
New episodic memory recorded for time 2.
Considering episode (time, cardinality, score) (2, 0, 0,000000)
Considering episode (time, cardinality, score) (1, 2, 2,000000)
NEW KING (perfect, graph-match): (false, false)
=>WM: (74: S1 ^operator O3)
    3: O: O3 (next)
--- apply phase ---
--- Firing Productions (PE) For State At Depth 1 ---
Firing epmem*apply*next
Retracting epmem*propose*next
New episodic memory recorded for time 3.

print --depth 10 e1
(E1 ^command C1 ^present-id 4 ^result R2)
 (C1 ^next N4)
 (R2 ^memory-id 2 ^present-id 4 ^retrieved R6 ^success N4)
  (R6 ^io I8 ^name epmem ^operator* O7 ^reward-link R7 ^superstate nil
       ^svs S9 ^type state)
   (I8 ^input-link I9 ^output-link O8)
   (O7 ^name next)
   (S9 ^command C5 ^spatial-scene S10)
    (S10 ^id world)
```

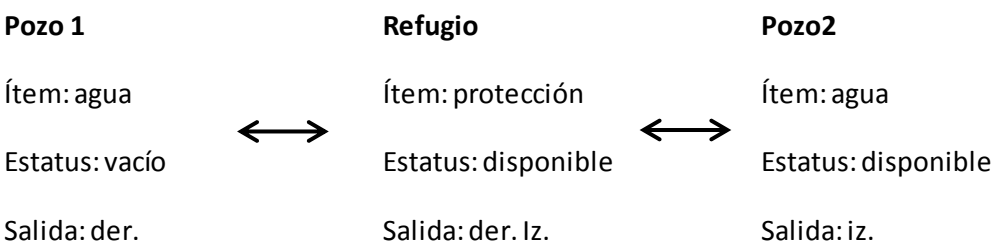
Agente con Episodic memory y Reinforcement Learning

Aquí vamos a ver un agente que hace uso a la vez de dos recursos de SOAR, aprendizaje y memoria, y saca partido de ambos para solucionar un problema y aprender de él. Va a ser el precursor del agente final. La teoría es la siguiente:

Pretendemos que este sistema capture automáticamente toda la experiencia, que sea capaz de aprender cuándo acceder a la Episodic memory, construir las consultas, cuándo usarlas, y para qué, teniendo en cuenta la estructura temporal de la memoria. Para aprenderlo usaremos recompensas RL.

El "Well world" es sencillo para el aprendizaje, pero suficientemente completo para que la memoria episódica haga su trabajo. La meta de un agente en este ambiente es satisfacer 2 necesidades, sed y seguridad. Existen 3 localizaciones, en dos de ellas

existen pozos, que proveen el agua, y en la tercera tenemos un refugio, que provee seguridad. El agente percibe sólo su localización actual. El entorno del problema es de la siguiente manera:



El agente se mueve entre las localizaciones y consume los recursos en ellas si están disponibles. La seguridad siempre esta disponibles pero el recurso de agua solo está disponible en un pozo cada vez. Si el agua está disponible en el pozo 1 y el agente la consume entonces se vacía y estará de nuevo disponible en el pozo 2. Así que el pozo que provea el agua alterna cada vez que el agente la consume.

El agente debería recordar donde bebió agua la última vez para saber qué pozo estará lleno después. Esta información podría ser guardada en la Working memory o en la Semantic memory, sin embargo esto requeriría unas reglas específicas que fueran guardando la información, pero la Episodic Memory lo hace automáticamente por lo que podemos aprovecharnos de ello. Lo interesante es que el agente no sabe que cuando bebe de un pozo no puede volver a beber y tampoco sabe que "tener sed" es una necesidad que se evita bebiendo agua, ni sabe sobre seguridad y refugio. El significado de esto debe ser aprendido gracias a las recompensas.

Cuando se satisface la necesidad de beber ésta se pone a 0, la sed se incrementa en 1 de forma lineal con cada "paso de tiempo". Cuando pasa de 10 el agente es considerado sediento hasta que satisfaga su sed consumiendo agua lo cual resetea la necesidad. La necesidad de seguridad es una necesidad constante y se satisface consumiendo el recurso en el refugio. Las acciones externas como moverse o consumir recursos se penalizan con una recompensa de -1, las acciones internas, como recuperar un episodio incurren en una recompensa de -0.1. El agente recibe una recompensa de -2 cada vez que da un paso y esté sediento y consigue una recompensa de +2 cuando consume el recurso de seguridad y no está sediento. Satisfacer la sed da una recompensa de 8. Se asume que pueden pasar cosas de forma concurrente.

El conocimiento inicial del agente consiste en los siguientes operadores, lo cuales son propuestos en cualquier estado: consumir recurso, moverse a una localización vecina y crear una consulta para iniciar una recuperación de memoria. El estado del agente incluye la localización del agente, los valores de sus necesidades y los resultados de

cualquier episodio recuperado. El agente además tiene una regla RL para cada posible estado, inicializadas todas a 0.0

Los resultados de las pruebas realizadas conforme a estas directrices muestran que el agente tarda aproximadamente 250 intentos en aprender el patrón más óptimo de comportamiento. El agente debe aprender que cuando tiene sed debe hacer una consulta de memoria, usando la consulta: "recurso: agua, disponible: si, acción: consumir" y entonces usar el conocimiento recuperado para aprender qué acción tomar después, la cuál será elegir moverse hacia el pozo contrario del cuál dice su memoria que consumió por última vez, ya que cada vez que se hace una recuperación de memoria se devuelve el estado que concuerde que sea más reciente. El agente del problema no puede percibir que pozos contienen agua mientras está en el refugio y debe usar la memoria para saber que ha ocurrido en el pasado. El comportamiento óptimo, entonces, es moverse al refugio y consumir el recurso de seguridad mientras no esté sediento. En el momento que lo está debe hacer la consulta anteriormente descrita.

Lo interesante de esto es que el agente podría ser capaz de aprender el comportamiento anterior sin necesidad de acceder a su memoria, solo a base de recompensas RL, sin embargo tardaría muchísimo más en llegar a la misma conclusión, por tanto es muy útil la utilización conjunta de la memoria de experiencias (Episodic memory) para acelerar el proceso. Esto demuestra la potencia de la combinación de 2 de las posibilidades que nos ofrece SOAR.

Vamos a ver en profundidad como se ha conseguido hacerle entender a la arquitectura que debe aprender ese comportamiento sin decirle directamente que debe hacerlo o cómo hacerlo. Para esto se presenta el código SOAR del agente, que se ejecuta desde un entorno Java que es el encargado de proporcionarle un contexto. En este problema la parte Java nos permite a través de la interfaz ver cuándo el agente tiene sed, donde está y que decide hacer en cada momento para nosotros poder analizar si está más o menos cerca del comportamiento esperado. El programa recibe de SOAR cada decisión que toma el agente y modifica los valores del contexto. Si se decide consumir un recurso, éste desaparece (por ejemplo con el agua del pozo, apareciendo en el pozo contrario), si decide moverse se le cambia todo el entorno para que pueda percibir lo que hay en su nueva ubicación, ya que el agente no debe tener capacidad para ver los pozos si está en el refugio o ver el refugio si está en un pozo etc…, así saca el máximo partido a su capacidad.

Comenzamos el código con nuestras reglas de inicialización que muestran las únicas variables que usa el agente, de las cuales es conocedor:

```
sp "propose*initialize
   (state <s> ^superstate  nil
```

```
        -^name)
-->
  (<s> ^operator <o> +)
  (<o> ^name initialize)
"
sp "apply*initialize
  (state <s> ^operator.name initialize)
-->
  (<s> ^count 0
       ^out 0                    ###Si out=1 muevel, si 2 consume, si 3 mueveD
       ^posicion r               ###Las opciones son refufio, pozo1, pozo2
       ^item ref                 ###Las opciones son ref y agua
       ^estatus 1                ###1 - disponible, 0 - no disponible
       ^sed 0                    ###Muestra la sed del agente
       ^actualizar 0
       ^proteccion 1
       ^mov N                    ###Control del sistema de RL y cambio de estado
       ^name well
       ^salida 2)                ###Muestra hacia donde se puede mover el agente.
"
```

Ahora se muestran los 3 operadores posibles que se pueden ejecutar en cualquier situación. La recuperación de un episodio de memoria, moverse y consumir el recurso si está disponible. Comenzamos con la recuperación de un episodio:

```
sp "epmem*propose*cueBased
  (state <s> ^name well
             ^actualizar 0
             ^mov N)
-->
  (<s> ^operator <op> +)
  (<op> ^name epMemCB)
"
sp "epmem*apply*cueBased
  (state <s> ^operator <op>
             ^actualizar <a>
             ^epmem.command <cmd>)
  (<op> ^name epMemCB)
-->
  (<s> ^actualizar <a> - 1)
  (<cmd> ^query <n1>)       ###Aqui empezamos a escribir la consulta a nuestra memoria
  (<n1> ^item agua          ###Nos interesa que encontremos la última vez que estuvimos en
        ^estatus 1          ###un sitio que había agua y estaba disponible, y la consumimos
        ^out 2              ###El hecho de saber si la llegamos a consumir lo sabremos en el
        ^operator* <op2>)   ###ciclo de flush (ciclo intermedio para funcionamiento de RL)
  (<op2> ^name flush)
"
```

Para el operador de movimiento las reglas son más complicadas. La regla de proposición de movimiento es simple pero existen varias reglas intermedias necesarias. Tenemos las reglas que modifican la recompensa recibida si el agente tiene sed y decide moverse, lo cual le recompensa negativamente, y tenemos las reglas que controlan si el movimiento elegido es adecuado (si se ha recuperado un episodio de memoria) dándole al agente una mayor recompensa si se dirige hacia el pozo contrario al último del que ha bebido. Por último tenemos las reglas normales de aplicación del

operador de movimiento que modifican la variable del output para que el entorno Java tenga el feedback de las decisiones. (Las reglas expuestas aquí están ligeramente modificadas para facilitar su comprensión. Las reglas reales están por duplicado, ya que no se puede gestionar el movimiento a la derecha o a la izquierda de forma sin ser de forma independiente.)

```
sp "propose*moverse
  (state <s> ^mov N)
-->
  (<s> ^operator <op> +)
  (<op> ^name mover)
"
sp "apply*moverse*SED
  (state <s> ^operator <op>
             ^reward-link <r>
             ^sed {<sd> >= 10})
  (<op> ^name mover)
-->
  (<r> ^reward.value -2)
"
sp "apply*moverse*epmem
  (state <s> ^operator <op>
             ^posicion r
             ^reward-link <r>
             ^epmem.result.retrieved <e1>)
  (<op> ^name mover)
  (<e1> ^posicion p1)              ###Si nos encontraramos en la situación de movernos a la Iz sería P2
-->
  (<r> ^reward.value 4)
"
sp "apply*moverse
  (state <s> ^operator <op>
             ^reward-link <r>
             ^count <c>
             ^sed <sd>
             ^out <o>)
  (<op> ^name mover)
-->
  (<r> ^reward.value -1)
  (<s> ^sed <sd> - (+ <sd> 1)
       ^mov N - CS
       ^out <o> - 3
       ^count <c> - (+ <c> 1))
"
```

Las reglas para consumir recursos son 4, una para proponer el operador, una para consumir agua de uno de los 2 pozos y 2 para gestionar el "consumo" del recurso refugio según si el agente tiene sed o no, lo cual incurre en diferentes recompensas.

```
sp "propose*consumir*recurso
  (state <s> ^estatus 1
             ^mov N)
-->
          (<s> ^operator <op> +)
          (<op> ^name consumir)
```

```
"
sp "apply*consumir*recurso*Agua
  (state <s> ^operator <op>
             ^count <c>
            ^sed <sd>
            ^out <o>
            ^item agua
            ^reward-link <r>)
  (<op> ^name consumir)
-->
  (<r> ^reward.value 8)
  (<s> ^sed <sd> - 0
     ^mov N - CS
     ^out <o> - 2
     ^count <c> - (+ <c> 1))
"
sp "apply*consumir*recurso*Ref
  (state <s> ^operator <op>
         ^count <c>
         ^sed {<sd> < 10}
         ^out <o>
         ^item ref
         ^reward-link <r>)
  (<op> ^name consumir)
-->
  (<r> ^reward.value 2)
  (<r> ^reward.value -1)
  (<s> ^count <c> - (+ <c> 1)
       ^sed <sd> - (+ <sd> 1)
       ^out <o> - 2
       ^mov N - CS)
"
sp "apply*consumir*recurso*RefS
  (state <s> ^operator <op>
             ^count <c>
             ^sed {<sd> >= 10}
             ^out <o>
            ^item ref
           ^reward-link <r>)
  (<op> ^name consumir)
-->
  (<r> ^reward.value -1)
   (<s> ^count <c> - (+ <c> 1)
        ^sed <sd> - (+ <sd> 1)
        ^out <o> - 2
       ^mov N - CS)
"
```

Posteriormente a los 3 operadores tenemos las reglas necesarias para que el sistema de RL funcione correctamente, lo cual ya se ha comentado en el apartado sobre RL. Hablamos de las reglas flush, que limpian nuestro registro de recompensas para evitar problemas derivados. Hemos aprovechado la presencia de estas reglas necesarias para utilizarlas más provechosamente, en este caso hemos añadido la gestión del input (cambio de situación) en ellas y además el borrado de los episodios recuperados con el

operador de recuperación de Episodic memory, ya que no es posible recuperar nuevas experiencias sin eliminar las anteriores.

```
sp "propose*flush*reward
   (state <s> ^name well
              ^mov CS)
-->
   (<s> ^operator <o> + )
   (<o> ^name flush)
"
sp "apply*flush*reward*2          ###Tenemos 2 aplicaciones del operador ya que una considera el
   (state <s> ^operator <o>       ###borrado de la episodic memory y la otra no, este código
              ^count <c>          ###corresponde a la primera, el 2º es similar.
              ^mov CS
              ^actualizar <a>
              -^epmem.command.<cmd>
              ^io.input-link <il>
                                  ^posicion <p>
                                  ^item <i>
                                  ^estatus <e>
                                  ^salida <sl>
                                  ^out <ou>
              ^reward-link <r>)
   (<r> ^reward <rw>)
   (<il> ^pos <iop>
         ^it <ioi>
         ^est <ioe>
         ^sal <iosl>)
   (<o> ^name flush)
-->
         (<s> ^count <c> - (+ <c> 1)
          ^mov CS - N
          ^out <ou> - 0
          ^actualizar <a> - 0
          ^salida <sl> - <iosl>
          ^estatus <e> - <ioe>
          ^item <i> - <ioi>
          ^posicion <p> - <iop> )
   (<r> ^reward <rw> -)"
```

Finalmente tenemos unas cuantas reglas de utilidad, necesarias. Las reglas RL, una regla de monitoreo del estado del agente y la regla de gestión de la salida para comunicación con el entorno Java. Es necesario diferenciar reglas RL cuando se tiene información del pasado de cuando no, ya que el comportamiento se basa precisamente en aprender (reforzar) las elecciones creadas cuando se dispone de ese conocimiento. Esto se comenta más en profundidad en el apartado de "Enseñar a una máquina a aprender" en (problemas encontrados y soluciones)

```
gp "rl*well*conEpmem
   (state <s> ^posicion [p1 r p2]
              ^name well
              ^epmem.command.<cmd>
              ^epmem.result.retrieved <e1>
              ^operator <o> +)
```

```
   (<e1> ^posicion [p1 r p2])
   (<o> ^name [consumir moverD moverI epMemCB])
-->
   (<s> ^operator <o> = 0)
"
gp "rl*well*sinEpmem
   (state <s> ^posicion [p1 r p2]
          ^name well
          -^epmem.command.<cmd>
          ^operator <o> +)
   (<o> ^name [consumir moverD moverI epMemCB])
-->
   (<s> ^operator <o> = 0)
"
sp "monitor*posicion
   (state <s> ^posicion <x>)
-->
   (write (crlf) |posicion = |<x>)
"
sp "output*well
   (state <s> ^io.output-link <ol>
              ^out <o>)
-->
   (<ol> ^pos.out <o>)"
```

Anexo F

Reglas de decisión del agente modelado

sp "propose*initialize

Si nos encontramos en el estado inicial y todavía no tiene nombre, proponemos el operador initialize.

sp "apply*initialize

Si el operador a aplicar es initialize, crea todas las entidades con los valores iniciales del problema.

sp "muerte*hambre

Si el atributo hambre alcanza el valor máximo, muere.

sp "muerte*sed

Si el atributo sed alcanza el valor máximo, muere.

sp "paso*final

*Si nos encontramos en el paso previo a paso*final2, propon el operador seguir.*

sp "paso*final2

Si el operador seguir ha sido propuesto, cambia de estado.

Sp "muerte*veneno

Si la comida ha sido envenenada, propon el operador muerte.

sp "muerte*aviso

Si el operador es muerte, avisa por la salida que el agente va a morir y cambia de estado.

sp "muerte*aplicación

Si nos encontramos en el estado previo a la muerte del agente, porpon el operador muerte2.

sp "muerte*aplicacion2

Si el operador muerte2 ha sido propuesto, cambia al estado final.

sp "muerte*final

Si nos encontramos en el estado final, muere.

sp "propose*consumir*recurso

Si el recurso esta disponible, proponer el operador consumir.

sp "apply*consumir*recurso*AguaS

Si se ha propuesto el operador consumir y la sed tiene un valor alto, el atributo sed vuelve a cero, aumentamos el hambre, se genera una salida externa con la acción y se cambia de estado para procesar una recompensa positiva.

sp "apply*consumir*recurso*AguaSN

Si se ha propuesto el operador consumir y la sed tiene un valor bajo, el atributo sed vuelve a cero, aumentamos el hambre, se genera una salida externa con la acción y se cambia de estado para procesar una recompensa negativa.

sp "apply*consumir*recurso*ComidaH

Si se ha propuesto el operador consumir y el hambre tiene un valor alto, se modifican el estado para comenzar la decisión de qué comer, se genera una salida externa con la acción y se procesará una recompensa positiva.

sp "apply*consumir*recurso*ComidaHN

Si se ha propuesto el operador consumir y el hambre tiene un valor bajo, se modifican el estado para comenzar la decisión de qué comer, se genera una salida externa con la acción y se procesará una recompensa negativa.

sp "apply*consumir*recurso*Ref

Si se ha propuesto el operador consumir y tanto el hambre como la sed tienen un valor bajo , se genera una salida externa con la acción y se cambia de estado para procesar una recompensa positiva.

sp "apply*consumir*recurso*RefSNC

Si se ha propuesto el operador consumir y el hambre a diferencia de la sed tienen un valor bajo, se genera una salida externa con la acción y se cambia de estado para procesar una recompensa negativa.

sp "apply*consumir*recurso*RefNSC

Si se ha propuesto el operador consumir y el hambre a diferencia de la sed tienen un valor alto, se genera una salida externa con la acción y se cambia de estado para procesar una recompensa negativa.

sp "apply*consumir*recurso*RefSC

Si se ha propuesto el operador consumir y tanto el hambre como la sed tienen un valor alto, se genera una salida externa con la acción y se cambia de estado para procesar una recompensa negativa.

sp "propose*Consumir*SumaX (A B C)

Si es posible coger una pieza entera de X, proponer el operador sumaX.

sp "propose*PartirX (A B C)

Si es posible partir una pieza que ya se tenga de X, proponer el operador partir.

sp "propose*QuitarX*Entero (A B C)

Si es posible dejar una pieza cogida previamente de X, proponer el operador quitarEntX.

sp "propose*QuitarX*Mitad (A B C)

Si es posible quitar la mitad de una pieza partida previamente de X, proponer el operador quitarMedX.

sp "apply*SumaX (A B C)

Si el operador sumaX ha sido propuesto, se añade la carga calórica del alimento X al conteo general de calorías.

sp "apply*PartirX (A B C)

Si el operador partirX ha sido propuesto, se resta la mitad de la carga calórica del alimento X al conteo general de calorías (el numero de alimentos de X cogidos no varía).

sp "apply*QuitarEntX (A B C)

Si el operador quitarEntX ha sido propuesto, se resta la totalidad de la carga calórica de un alimento X al conteo general de calorías.

sp "apply*QuitarMedX (A B C)

Si el operador quitarMedX ha sido propuesto, se resta la mitad de la carga calórica del alimento X al conteo general de calorías (el número de alimentos de X cogidos se reduce).

sp "propose*AplicarHambre

Si el estado general es previo a la decisión de comer, proponer operador ApHambre.

sp "apply*AplicarHambre2

Si el operador ApHambre ha sido propuesto y el atributo hambre esta por debajo de cierto umbral, fijar el valor de la cantidad de calorías necesarias para saciarla.

sp "apply*AplicarHambre1

Si el operador ApHambre ha sido propuesto y el atributo hambre esta por encima de cierto umbral, fijar el valor de la cantidad de calorías necesarias para saciarla.

sp "propose*Contar

Si se ha decidido ya que comer, se propone el operador contar

sp "apply*Contar

Si se ha propuesto el operador contar, reiniciar todos los atributos presentes en la elección de comida y generar una salida con la decisión final tomada.

sp "propose*moverseX(D I S N)

Si el movimiento hacia X está disponible, proponer moverse hacia X con el operador moverX.

sp "apply*moverseX(D I S N)

Si el operador moverX ha sido propuesto, generar una salida indicando la acción de movimiento.

sp "epmem*propose*cueBased

Si es posible consultar la memoria episódica, proponer el operador epmem.

sp "epmem*apply*cueBased

Si el operador epmem ha sido propuesto, hacer una consulta a la memoria episódica de cuándo se bebió por última vez de un pozo.

sp "propose*SMretrieval1

Si el estado general es previo a la decisión de comer, proponer el operador smem.

sp "apply*SMretrieval1

Si el operador smem ha sido propuesto, consultar en la memoria semántica los datos de la primera comida ofrecida en la zona de alimentación. Cambiar de estado para guardar la información.

sp "propose*GuardarYLimpiar

Si se ha cambiado de estado para guardar la información de la semantic memory, proponer el operador flushSM

sp "apply*GuardarYLimpiar

Si se ha propuesto el operador flushSM se copia toda la información descargada de la memoria semántica en los atributos del estado principal y se cambia de estado a la consulta de la segunda comida ofrecida.

sp "propose*SMretrieval2

Si el estado es la consulta de la segunda comida ofrecida, proponer el operador smem2.

sp "apply*SMretrieval2

Si el operador smem2 ha sido propuesto, consultar en la memoria semántica los datos de la segunda comida ofrecida en la zona de alimentación. Cambiar de estado para guardar la información.

sp "propose*GuardarYLimpiar2

Si se ha cambiado de estado para guardar la información de la memoria semántica, proponer el opedaor flushSM2.

sp "apply*GuardarYLimpiar2

Si se ha propuesto el operador flushSM2 se copia toda la información descargada de la memoria semántica en los atributos del estado principal y se cambia de estado al comienzo de la decisión de qué comer.

sp "apply*moverseX*SED (I D)

Si se ha propuesto el operador moverX y el atributo sed esta por encima de un umbral, crea una recompensa positiva.

sp "apply*moverseX*Hambre (S N)

Si se ha propuesto el operador moverX y el atributo hambre está por encima de un umbral, crea una recompensa positiva.

sp "apply*moverseX*HambreN (S N)

Si se ha propuesto el operador moverX y el atributo hambre está por debajo de un umbral, crea una recompensa negativa.

sp "apply*moverseD*epmem*Patron2

Si se ha propuesto el operador moverD y la posición es adecuada al patrón 2, crea una recompensa positiva.

sp "apply*moverseD*epmem*Patron3

Si se ha propuesto el operador moverD y la posición es adecuada al patrón 3, crea una recompensa positiva.

sp "apply*moverseD*epmem*Patron1

Si se ha propuesto el operador moverD y la posición es adecuada al patrón 1, crea una recompensa positiva.

sp "apply*moverseI*epmem*Patron1

Si se ha propuesto el operador moverI y la posición es adecuada al patrón 1, crea una recompensa positiva.

sp "propose*flush*reward

Si el estado es recuperar y limpiar las recompensas, porponer el operador flush.

sp "apply*flush*reward*1

Si el operador flush ha sido propuesto y acaba de ocurrir una petición a la memoria episódica, eliminar todos los atributos de recompensas y todos los atributos de la memoria episódica. Cambiar los atributos del entorno si es necesario, comparándolos con los datos de entrada.

sp "apply*flush*reward*2

Si el operador flush ha sido propuest , eliminar todos los atributos de recompensas. Cambiar los atributos del entorno si es necesario, comparándolos con los datos de entrada.

sp "propose*modificar*umbralC

Si el estado de verificación de los umbrales de hambre y sed ha sido seleccionado, proponer el operador umbral.

sp "apply*modificar*umbralCX [bis]

Si el operador umbral ha sido seleccionado y dependiendo de la los valores de los atributos hambre y sed y de la presencia de recompensas, modificar los atributos de umbrales de sed y hambre y eliminar, en el caso de que las haya, recompensas redundantes.

sp {water-jug*evaluate*state*success

Si la cantidad de calorías elegidas para consumir concuerda con la cantidad de calorías seleccionadas entre todos los productos escogidos, crea un estado de éxito.

sp {water-jug*evaluate*state*failure

Si es mismo estado de elección de productos ha sido alcanzado por segunda vez, crea un estado de fallo para evitar continuar por ese camino, evitando ciclos.

sp {Impasse__Operator_Tie*elaborate*superstate-set

Si nos encontramos en un sub-estado derivado de un impasse del tipo operador tie, apunta el subestado al inmediatamente superior.

sp {Impasse__Operator_Tie*elaborate*superstate-set2

Si nos encontramos en un sub-estado derivado de un impasse del tipo operador tie, apunta el subestado al inmediatamente superiorde segundo nivel.

sp {water-jug*elaborate*problem-space

Si nos encontramos en un nuevo subestado copia todos los valores del problema-space.

sp "output*sed

Si hay algún cambio en el atributo sed, genera una salida con el nuevo valor.

sp "output*hambre

Si hay algún cambio en el atributo hambre, genera una salida con el nuevo valor.

sp "output*well

Si hay algún cambio en el atributo out, genera una salida con el nuevo valor.

Reglas combinatorias para el aprendizaje por refuerzo:

gp "rl*well*conEpmem

```
   (state <s> ^posicion [p1 r p2 c]
            ^umbralC [ 0 1 ]
                ^umbralS [ 0 1 ]
            ^name well
            ^epmem.command.<cmd>
            ^epmem.result.retrieved <e1>
            ^operator <o> +)
   (<e1> ^posicion [p1 r p2 c])
   (<o> ^name [consumir moverD moverI moverN moverS epMemCB])
-->
   (<s> ^operator <o> = 1)
"
```

gp "rl*well*sinEpmem

```
   (state <s> ^posicion [p1 r p2 c]
                ^umbralC [ 0 1 ]
                ^umbralS [ 0 1 ]
            ^name well
            -^epmem.command.<cmd>
            ^operator <o> +)
   (<o> ^name [consumir moverD moverI moverN moverS epMemCB])
-->
   (<s> ^operator <o> = 1)
"
```

Anexo G

Problemas encontrados y su resolución

Problema con la gestión del input en JSoar

Cuando crear los primeros agentes conectando SOAR con Java lo hacemos por la utilidad de poder pasar parámetros al agente desde el exterior, y que éste sea capaz de reaccionar ante esos estímulos y tomar decisiones con algo más que su conocimiento interno de sus reglas.

El problema para su correcta implementación aparece porque se gestionan los inputs con un Listener, una interrupción que salta cada vez que SOAR está en fase de input (primera fase de su ciclo) y que debemos gestionar con un procedimiento (rutina) escrita en Java. Como la primera vez que se ejecuta el agente se deben crear las estructuras que queremos "enviar" o "construir" en el input-link desde Java tenemos que diferenciar una ejecución diferente para la primera vez que se ejecuta la rutina, que para todas las siguientes. Para esto se ha utilizado la técnica del uso de un valor que se inicia a True y que se modifica a False inmediatamente al terminar la primera ejecución, a partir de la cual se ejecutará la rutina normal.

A parte de eso, nos encontramos con el problema mayor, y es que al crearse por primera vez la estructura del input-link los procedimientos de creación devuelven un valor de tipo InputWme con el "puntero" a esa parte de la memoria de SOAR. Cuando queremos modificar alguno de los valores del input, necesitamos ese puntero, pero no es posible recuperarlo más allá de conservar el puntero devuelto al crear cada valor.

Por lo tanto la única solución posible es conservar cada valor del tipo InputWme como variable privada global a todos los Listeners de esa clase Java, para poder acceder a ellos en cada rutina de interrupción de input. Al crearlos de forma privada y ser gestionados sólo en la rutina input no corre peligro su integridad.

Problema con la gestión del output en JSoar

De la misma manera que con los input, la utilidad de poder recoger información del agente SOAR hacia el exterior es obvia, ya que no solo la información sino las acciones que el pudiera tomar si se ejecutara sobre un robot real o un sistema mecánico modificarían su entorno y su situación.

El problema surgido para gestionar el output es que en los pocos y sencillos ejemplos que se pueden conseguir de SOAR la línea de output sólo gestiona un valor único de tipo simple que se devuelve con un handler a una clase java que solo contiene un valor entero público. Sin embargo al intentar gestionar 2 valores de output nos encontramos con algunos problemas.

El problema inicial fue como poner los valores en SOAR en la rutina de output para que el handler supiera sacarlos. La resolución a esto fue darse cuenta de que debían colgar ambos del mismo atributo y con el mismo nombre que el de las variables de la clase en Java, lo cual no es demasiado trivial. Pero el problema posterior, más grave, es que debido a que se gestionan dos valores de output, estos valores la arquitectura no los gestiona a la vez, sino que crea dos interrupciones, una por cada valor, por lo que hay que tener mucho cuidado al modificar los valores en el programa principal para que no haya errores, ya que además el orden de las interrupciones es aleatorio. A veces el valor de X entra primero y a veces segundo, acompañado con un 0 en el otro valor, al igual que Y, por lo que hay que evitar que X o Y puedan tener el valor 0 en ningún momento para poder filtrar bien que interrupción se está tratando en cada momento.

Problema de desbordamiento con el uso de Reinforcement Learning

El problema más grave enfrentado hasta ahora ha sido la gestión de las recompensas en el uso de RL.

Al consultar todos los ejemplos disponibles sobre uso de RL en SOAR nos encontramos que todos utilizan el sistema de dejar que el agente funcione de forma básica y asignarle una recompensa cuando llega al final. Tras lo cual el agente muere y se debe volver a iniciar conservando los datos de RL cada vez. Finalmente tras unos cuantos intentos el agente es capaz de hacer su elección casi de una forma óptima. Sin embargo nuestra forma de utilizar el RL no puede ser así, ya que queremos que el agente sea capaz de ir aprendiendo dinámicamente sin matarlo tras cada recompensa conseguida.

La implementación de esto parece trivial, ya que simplemente escribiendo las reglas tal como se ha mostrado en el apartado anterior (Reinforcement Learning) SOAR debería gestionar cada recompensa, modificar los valores y continuar. Sin embargo si lo programamos así vemos que la realidad es que cada vez que una recompensa nueva es dad al agente en vez de sobrescribir el valor de la anterior recompensa se añade a la estructura de reward-link con dos problemas en su funcionamiento, por un lado que cuando se modifican los porcentajes y existe más de un valor en reward-link todos los que hay son sumados y añadidos y son determinantes para modificar el porcentaje de la decisión tomada, lo cual es erróneo ya que cada valor de recompensa corresponde a una elección de operador concreta y no deben ser acumulables. Y por otro lado la estructura de reward-link con el paso de los ciclos va llenándose provocando finalmente un desbordamiento de pila. Por lo tanto era inviable la gestión de esa manera.

En un principio para intentar evitar la acumulación y el desbordamiento se creó una regla del tipo “elaboration” que se ejecutan en cada ciclo que borraba el reward-link tras haber puesto el valor de la recompensa al elegir un operador, sin embargo el

sistema RL no funcionaba y gestionaba cada recompensa con un valor 0. Por lo tanto esta solución no era correcta.

Tras esto, e intentar gestionar un sistema más simple de RL en el cual se diera la recompensa al final se vio que era necesario tener un ciclo adicional desde que se da la recompensa al final hasta que puedes destruir el agente, ya que sino tampoco era recuperado el valor de ésta. Por lo que se llegó a la conclusión de que era necesario un ciclo entero adicional tras cada recompensa ofrecida antes de poder borrar el valor del reward.link. Por eso se añadieron las siguientes reglas al agente creado en el apartado de Reinforcement Learning:

```
sp "propose*flush*reward
   (state <s> ^count <c>
          ^name cont
          ^mov S)
-->
   (<s> ^operator <o> + )
   (<o> ^name flush)"
sp "apply*flush*reward
   (state <s> ^operator <o>
          ^count <c>
          ^mov S
          ^reward-link <r>)
   (<r> ^reward <rw>)
   (<o> ^name flush)
-->
  (<s> ^count <c> - (+ <c> 1))
   (<s> ^mov S - N)
   (<r> ^reward <rw> -)"
```

La estructura de estas reglas es la de proponer y aplicar un operador, como si fuera una decisión normal del agente, sin embargo aquí entra el juego el valor "^mov" que le habíamos dado al agente al inicio y que, en principio, no tenía utilidad. Este valor va a hacer de "interruptor" y obligar a que tras cada decisión de elección de movimiento haya un ciclo "vacío" el cual se utilizará para dar tiempo suficiente al sistema de RL a recoger la recompensa y borrarla del reward-link para evitar el overflow de la pila. Por eso éste operador solo será propuesto cuando "^mov" tenga el valor "S" y creará un operador único, cuya aplicación eliminará la recompensa previa: *(<r> ^reward <rw> -)".* Y Modificará de nuevo el parámetro ^mov a N, que es el funcionamiento normal del agente: *(<s> ^mov S - N).*

Con esto, añadimos un ciclo adicional cada elección, pero debido a su sencillez el tiempo que tarda SOAR en pasar este ciclo es muy pequeño y no intercede en el funcionamiento que queremos que tenga el agente, permitiendo el uso del RL que de otra forma es imposible de implementar bien.

Nos encontramos con el mismo problema en la meta del problema, donde necesitamos un ciclo adicional antes de terminar para que la recompensa por conseguir el objetivo se vea reflejada, usamos 2 reglas similares a las anteriores más una final:

```
sp "propose*flush*reward*fin
   (state <s> ^count <c>
            ^name cont
            ^mov F)
-->
   (<s> ^operator <o> + )
   (<o> ^name flushFin)
"
sp "apply*flush*reward*fin
   (state <s> ^operator <o>
            ^count <c>
            ^mov F
            ^reward-link  <r>)
   (<r> ^reward <rw>)
   (<o> ^name flushFin)
-->
            (<s> ^count <c> - (+ <c> 1))
   (<s> ^mov F - E)
   (<r> ^reward <rw> -)
"
sp "fin
   (state <s> ^count <c>
            ^name cont
            ^mov E)
-->
   (halt)
```

"**Muerte de un agente tras alcanzar su meta**

En el momento en el que se propuso utilizar el método del Chunking, que funciona a través del sistema de selección de operadores cuando nos encontramos un impasse "operator tie" se vio que existía un impedimento difícil de superar, y es que los pocos ejemplos que se pueden consultar de éste sistema están basados en problemas de lógica con una finalidad concreta tras la cual ya han cumplido su cometido y acaban su ejecución.

Esto significa que el agente empezaba directamente con un "operator tie" y se hacía una simulación de todo el problema propio que se le presentaba con todas sus posibilidades hasta llegar al final, hacer backtracking y usar Chunking para aprender el camino. Tras eso moría.

Sin embargo el agente que se pretende crear para este proyecto debe sobrevivir a una simulación y al uso del Chunking para aprender y seguir con su "vida".

Cuando se crea la entidad "desire" (ver Anexo 2 Chunking) con el estado final que se desea y la regla que comprueba que se ha llegado a ese final no se puede evitar que,

cuando el backtracking del agente llega al estado inicial, éste piense que ya ha terminado totalmente y aun en la ausencia de una instrucción “halt” (finalizar agente) éste termina su ejecución.

Al principio se pensó que esto era un error propio de programación, pero tras consultar cierta documentación se vió que este comportamiento estaba creado específicamente par que funcionara de esta manera, lo cual impedía a nuestro agente funcionar debidamente. Por ello tuvimos que encontrar la manera de evitar su muerte prematura.

La única solución viable fue evitar que la función de final satisfactorio, la regla que permite el backtracking cuando se ha llegado a la meta, se ejecutara en el estado inicial, forzándola a parar. De esta manera el backtracking funciona perfectamente hasta el estado previo al inicial y en éste se crea una regla especial que evita la muerte saltándose el último paso. El “chunk” se crea igualmente y el funcionamiento del agente prosigue de forma normal.