Universidad Zaragoza

Proyecto Fin de Carrera

Creación de un Agente basado en la arquitectura SOAR

Autora

Teresa Zoe Albajar Lafarga

Directores

Dr. Francisco José Serón

Dr. Manuel G. Bedía

Departamento de Informática e Ingeniería de Sistemas

Escuela de Ingeniería y Arquitectura

2015

Creación de un Agente basado en la arquitectura SOAR

Resumen

Dentro del conjunto de arquitecturas cognitivas existentes, la arquitectura SOAR nos proporciona los módulos necesarios para crear agentes dotados de una serie de características que podríamos considerar "inteligentes".

El objetivo de este proyecto es estudiar a fondo la arquitectura SOAR, analizar su alcance y utilizarla.

Esta arquitectura utiliza un sistema basado en reglas que permiten tomar decisiones y crear un comportamiento.

Se presenta un problema para el que se crea un agente que interactúa con el entorno. Debe tomar decisiones para intentar sobrevivir el máximo tiempo posible y aprender de ellas para optimizar sus acciones. El agente creado para ésta arquitectura hace uso de los módulos que SOAR ofrece para realizar dos tipos de aprendizaje: uno a largo plazo y otro a corto plazo.

El entorno del problema se simula con un programa desarrollado en Java que se comunica con el agente (ejecutado en la arquitectura SOAR). A través de una interfaz gráfica se controla al agente y se muestran las decisiones tomadas por él.

Se han analizado las diferentes librerías que se ofrecen para comunicar el sistema Java con la arquitectura SOAR y se ha seleccionado una de ellas para esta implementación.

Finalmente se presentan las conclusiones obtenidas a partir de los resultados y el posible trabajo futuro.

Agradecimientos

A mi familia y amigos, por toda su ayuda y su comprensión sobre todo en los momentos más difíciles.

A Carlos, por su apoyo incondicional y por creer en mí incluso cuando yo misma no pude. Por enseñarme a ver que no todo es blanco o negro y regalarme a diario una vida más feliz.

Y por último a la memoria de mis abuelos Luis Lafarga y Fernando Albajar, y de mi tío abuelo José Mª Lafarga, quien me enseñó que la verdadera fuerza es la voluntad.

INDICE

PARTE 1

MEMORIA

1. Introducción

1.1 Sistemas Cognitivos

Una arquitectura cognitiva define las estructuras que conforman los módulos con los que es posible crear unos sistemas que poseen características que podríamos denominar "inteligentes" en entornos concretos.

Uno de los retos fundamentales en el desarrollo de agentes con aspectos que simulan a un humano es definir las estructuras computacionales y de organización que permitan guardar, recuperar y procesar el conocimiento adquirido. Igualmente importante es definir cómo están organizadas estas estructuras. En un computador estándar, la arquitectura incluye un **diseño** total de:

- La **memoria**, en la que se almacenan los datos y las instrucciones.
- El **control de los componentes de procesado**, que ejecutan las instrucciones de los programas basados en operaciones primitivas, moviendo datos entre memorias y determinando el orden de las instrucciones. El lenguaje debe ser general y flexible para que pueda ser implementado eficientemente y sea confiable.
- Las diferentes representaciones de los **datos**.
- Los **dispositivos de entrada y salida**.

Las arquitecturas cognitivas están ideadas para crear agentes generales y autónomos que sean capaces de resolver una gran diversidad de tareas utilizando variedad de conocimientos. Una de las primeras dificultades a la que se enfrentan las arquitecturas cognitivas es la de coordinar entre sí todas las capacidades que normalmente consideramos que poseen los sistemas inteligentes tales como la percepción, la interacción, el razonamiento, la planificación, la toma de decisiones, el procesamiento del lenguaje, el aprendizaje o la codificación del conocimiento entre otros, y a ser posible en entornos dinámicos.

1.2 SOAR como sistema cognitivo

1.2.1 Glosario

Entidad: Variable interna que consta de un identificador y múltiples atributos. No tiene una estructura fija. Cada atributo con nombre único tiene un valor, que puede ser de tipo entero, carácter u otra entidad.

Estado: Situación en la que se encuentra un sistema en un momento concreto en relación a los valores de los atributos de sus entidades.

Regla de decisión: Código que consta de 2 partes. La *parte izquierda* contiene las condiciones necesarias para que el código pueda ejecutarse. Estas condiciones serán comparadas con el *estado* del sistema. La *parte derecha* consta de una serie de acciones que modifican las entidades del sistema, afectando de esta forma al *estado*.

Agente: Es el conjunto de *reglas de decisión* que se ejecutan en un entorno concreto generando un comportamiento.

Operador: Tipo especial de entidad que hace referencia a una regla que ejecuta una serie de acciones. Posee atributos como el "nombre" y el "peso" que sirven para marcar diferencias entre los operadores.

Problem space: Situación general del entorno y del propio agente para resolver un problema concreto utilizando los medios y la información de las entidades a las que tiene acceso.

Meta: Objetivo que se quiere alcanzar en el problema planteado al agente, considerando el *problem space* en el que se encuentre.

1.2.2 Descripción

SOAR son las siglas de: *estado* (State), *operador* (Operator) y resultado (And Result), que reflejan una arquitectura cognitiva diseñada con el fin de resolver diversos tipos de problemas. Esta arquitectura utiliza un sistema basado en reglas, a partir de las cuales se toman decisiones. Estas decisiones se ejecutan al aplicar *operadores* a un *estado* de un agente que tiene como objetivo alcanzar una meta [4].

La mayoría de los métodos ofrecidos por la Inteligencia Artificial se diseñan para resolver una tarea concreta, en la cual se vuelven muy diestros (por ejemplo jugar al ajedrez). Sin embargo la arquitectura SOAR trata de proporcionar los medios para que los *agentes* sean capaces de realizar de forma óptima **multitud de tareas diferentes** en un entorno dinámico y **adaptarse a los imprevistos del entorno** por sí mismos.

Las arquitecturas cognitivas aportan estructuras computacionales representadas por módulos que se relacionan entre sí. Estas relaciones permiten la creación de sistemas de mayor complejidad.

Esta arquitectura cognitiva se enfoca a la consecución de *metas* dentro del espacio específico de un problema (*problem space*); para ello utiliza *reglas de decisión,* que trabajan a su vez con *estados* y *operadores*. En la Fig. 1 se observa la estructura de la arquitectura.

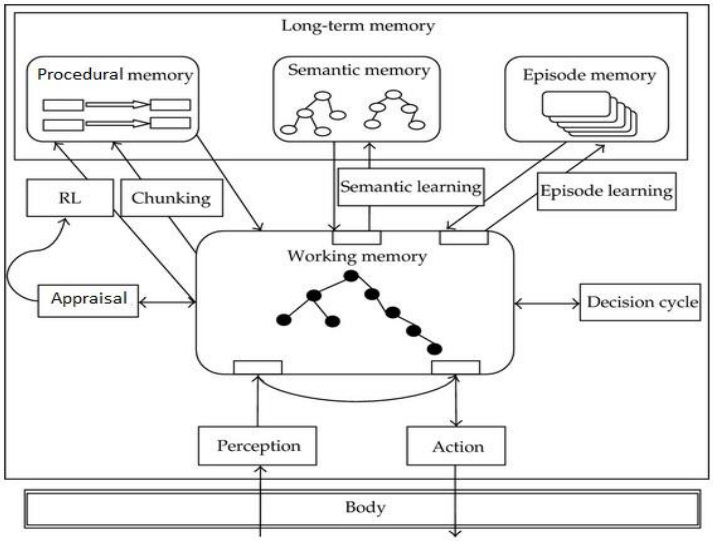

Figura 1 - Diagrama de bloques de SOAR [1].

Esta vista estructural de SOAR nos muestra un bloque superior que engloba todos los módulos que conforman el sistema de toma de decisiones, y un bloque con doble borde en la parte inferior (*Body*), que representa la parte "física" del sistema.

El bloque inferior interacciona con el sistema de toma de decisiones transmitiéndole información acerca de su entorno y lleva a cabo las acciones derivadas de las decisiones tomadas por el sistema modificando así dicho entorno. La relación entre ambos bloques se asemeja a la de un cuerpo en colaboración con el cerebro.

Los módulos rectangulares de la fig. 1 representan los diferentes procesos de aprendizaje y toma de decisiones, los módulos con esquinas redondeadas muestran las diferentes memorias disponibles.

A partir de los datos recibidos del exterior por los diferentes sensores de los que se disponga, el *estado* en el que se encuentra el agente se irá modificando. Esto sucede durante un ***ciclo de ejecución*** (ver Fig. 2)

Podemos **definir** un *ciclo de ejecución* como los pasos que se realizan durante el tiempo que transcurre desde que se empiezan a **procesar los datos recibidos** del entorno hasta que se **toma una decisión** y se **envía una respuesta** que contiene las acciones a realizar.

Los ciclos se van sucediendo en el agente hasta alcanzar un *estado* que sea equivalente a la *meta* especificada para ese problema en particular.

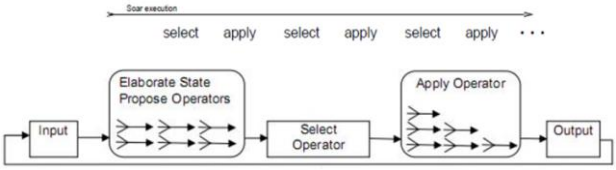

Figura 2 – Diagrama del ciclo de ejecución de SOAR [2].

Dentro de un ciclo de ejecución, además de la gestión de los datos de entrada y salida con el entorno existen tres fases:

1. Preselección de los *operadores* para este ciclo.
2. Selección del *operador* a aplicar (el método de selección se detalla en el punto 2.1).
3. Aplicación del *operador* elegido modificando los atributos de las entidades que intervengan. Se generan las salidas (respuestas) hacia el entorno en forma de información o acciones.

1.3 Objetivos del proyecto

El objetivo principal de este proyecto es estudiar e investigar la arquitectura cognitiva SOAR y utilizarla para modelar el comportamiento de un agente que evolucione con el paso del tiempo. Se pretende que el agente haga uso de todos los sistemas principales de la arquitectura SOAR.

Es decir, que el agente podrá utilizar su conocimiento previo y su conocimiento adquirido en el tiempo para aprender a optimizar sus acciones de forma que sea capaz de sobrevivir el máximo tiempo posible.

El agente tendrá definidas un conjunto de necesidades que deberá atender y que estarán relacionadas con el hambre y la sed. Tendrá que intentar cubrir estas necesidades para conseguir su meta de supervivencia manteniéndose la mayor parte del tiempo en zona segura.

El problema propuesto al agente está planteado para poder explorar todos los tipos de memoria y aprendizaje que ofrece la arquitectura SOAR.

Además del sistema SOAR se ha implementado una aplicación Java que utiliza librerías específicas para crear la conexión con la arquitectura SOAR. Dado que existen dos conjuntos de librerías diferentes que se pueden utilizar para este propósito, se realizará una fase previa de experimentación con ambas y se obtendrán conclusiones sobre la idoneidad de uso de una de ellas para resolver el problema planteado.

1.4 Motivación

- Ahondar en el estudio de la Inteligencia Artificial utilizando esta arquitectura que es diferente a las estudiadas durante la carrera.
- Desarrollar un sistema capaz de aprender y reaccionar a su entorno.
- Estudiar de forma general las limitaciones del sistema en vistas a trabajos futuros con SOAR.

1.5 Estructura de la memoria

En la primera parte de la memoria se presenta una introducción a este proyecto, en la segunda se plantea un resumen del trabajo de investigación sobre SOAR, en la tercera y cuarta parte se aborda el problema y se presentan los resultados y conclusiones:

- **1ª parte:** se expresa de forma breve qué es un sistema cognitivo, para posteriormente introducir la arquitectura SOAR. Se definen los objetivos del proyecto que se pretenden alcanzar y la motivación que llevó a su desarrollo.
- **2ª parte:** se divide en dos apartados. En primer lugar se hace un resumen detallado de la arquitectura SOAR (explicando sus diferentes módulos y funcionamiento básico) y posteriormente se presentan las librerías para la implementación de SOAR con las ventajas e inconvenientes de la utilización de cada una de ellas.
- **3ª parte:** se estructura en dos apartados: En el primero se presenta el problema modelado y su implementación, y en el segundo se muestran los resultados obtenidos y se explica la interfaz desarrollada.
- **4ª parte:** se muestran las conclusiones y se presenta el posible trabajo futuro.

2. SOAR

2.1 Presentación en detalle

La memoria de trabajo

La arquitectura SOAR basa todo su funcionamiento en el uso de una **memoria a corto plazo** llamada *memoria de trabajo* (Working Memory). Se suele representar en el centro de los módulos que conforman la arquitectura (ver Fig.1, página 8) ya que está conectada con todos ellos. En la *memoria de trabajo* se almacena el *estado* actual del *agente* (conforme se modifica) y la *meta* que debe alcanzar.

Todo el contenido de la *memoria de trabajo* se codifica de forma simbólica. Esta codificación utiliza nodos, hojas y uniones entre ellos (ver Fig. 3).

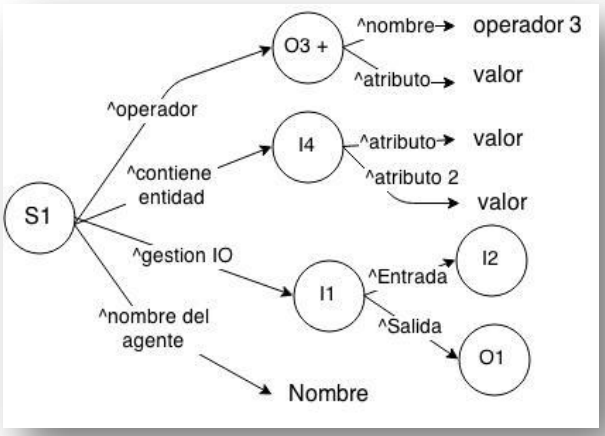

Figura 3 – Codificación Simbólica de un estado SOAR

La sintaxis utilizada tiene una serie de características:

- En esta codificación los **círculos** representan los nodos, conteniendo el identificador de una *entidad*. Estos identificadores suelen ser combinaciones de una letra y un número, elegido por la arquitectura en el momento de crear la *entidad*. El identificador asignado a una entidad no tiene relevancia en ella, solo funciona como puntero a la hora de modificar sus atributos.
- Los arcos de unión poseen un identificador que se corresponde con nombre del atributo al que apuntan (ver Fig. 3). El identificador del arco de unión se diferencia de otros identificadores porque tiene "^" como carácter inicial.
- Este atributo al que apuntan los arcos puede ser otro nodo o un valor (numérico o carácter), en cuyo caso se representa sin utilizar un círculo.

Por ejemplo, en la Fig. 3 se observa una entidad con un identificador "S1" con 4 atributos (^operador, ^contiene entidad, ^gestión IO y ^nombre del agente). El atributo "^operador" apunta a otro nodo: otra *entidad* llamada "O3+", que posee otros dos atributos (^nombre y ^atributo) con sus valores ("operador 3" y "valor").

En la arquitectura SOAR existen algunos **identificadores** de entidades y de atributos **ya establecidos** que no deben modificar su nombre debido a que diferentes sistemas hacen uso de ellos. Estos identificadores y atributos están presentes en el *estado* de cualquier *agente* (ver fig. 4).

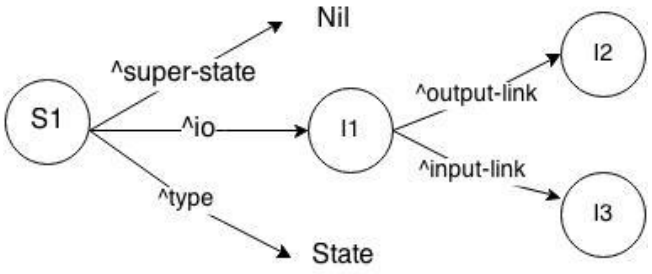

Figura 4 – estado básico de un agente

El nodo inicial del *estado* de cualquier *agente* tiene el identificador "S1". Este nodo posee tres atributos:

- Un atributo ^type (tipo), cuyo valor es "State" (estado).
- Un atributo ^super-state cuyo valor en el *estado* principal es "Nil".
- Un atributo ^io (entrada-salida) el cual tiene como valor otra entidad, "I1" que se divide en otros 2 atributos (uno de entrada y otro de salida), los cuales apuntan a las entidades "I2" e "I3", que se utilizan para intercambiar información con el entorno.

La memoria procedural

Además de la *memoria de trabajo*, SOAR ofrece también memoria a largo plazo. Como se observa en la Fig. 1 (página 6) la arquitectura ofrece tres módulos de memoria de éste tipo. La más importante (e imprescindible) de ellas es la *memoria procedural* (Procedural memory o Production memory). Está unida a la *memoria de trabajo* a través de otros módulos (que se describirán en el punto 2.1.1).

La *memoria procedural* contiene las *reglas de decisión*[1] del agente. Estas reglas son la base del comportamiento ya que condicionan cada decisión tomada por el agente.

[1] Desarrollado en el Anexo A: Programación de Reglas.

Las *reglas de decisión* pueden ser escritas por el programador del *agente* o pueden ser creadas por la propia arquitectura.

Cuando están escritas por el programador se cargan en la *memoria procedural* antes del comienzo de la ejecución. Sin embargo, cuando las reglas son creadas por la propia arquitectura se almacenan automáticamente en la memoria. Estas reglas se crean a partir de unos patrones codificados en reglas especiales escritas por el programador. Debido a que los patrones se basan en **combinar** distintos valores de ciertas entidades, el número de reglas a crear es un número combinatorio en función de la cantidad de entidades y valores que estas pueden tomar, y por eso son denominadas *reglas combinatorias*.

Como se ha presentado en el glosario de términos, una *regla de decisión* tiene dos partes separadas: la primera, llamada *parte izquierda*, contiene las *condiciones;* la segunda, *parte derecha*, contiene las *acciones*.

Las *condiciones* de la *regla de decisión* están formadas por el mismo tipo de combinaciones de *entidad, atributo y valor* que el *estado* del agente. Si las *condiciones* de la *regla de decisión* **enlazan** con el *estado* del agente entonces se ejecutan las *acciones* de la misma. Existen tres tipos diferentes de reglas según las *acciones* que posean:

- En el **primer tipo** de reglas las *acciones* están dirigidas a **proponer *operadores***.
- Estos *operadores* (como se presenta en el Glosario), enlazan con el **segundo tipo** de reglas cuyas *acciones* consisten en **modificar los atributos** de algunas *entidades.*
- Y el **tercer tipo** de reglas poseen *acciones* referentes al **intercambio de datos** con el entorno que son tratadas de forma independiente.

El módulo de decisión

En el *ciclo de ejecución* de SOAR (ver Fig. 2 Bis) se enlaza el *estado* del *agente,* almacenado en la *memoria de trabajo,* con las *condiciones* de las *reglas de decisión* y se ejecutan las *acciones* de las *reglas* cuyas *condiciones* se cumplan en dicho *estado*. En el ciclo de ejecución se distinguen varios pasos:

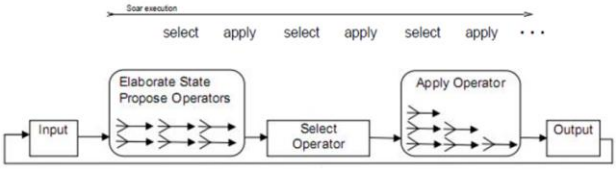

Figura 2 Bis – Ciclo de ejecución de SOAR [2].

1. El *ciclo de ejecución* comienza con la gestión de entrada de información desde el entorno. Esta información es añadida al *estado* del agente en una *entidad* con un identificador fijo "I2" (ver Figura 4, página 14).
2. El segundo paso es la pre-selección de *operadores*. Como se ha presentado previamente existe un tipo de *regla de decisión* cuyas *acciones* son la proposición de *operadores*. En este paso del ciclo las *reglas* cuyas *condiciones* se cumplan proponen un determinado número de *operadores,* que se almacenan para el siguiente paso.
3. El tercer paso precisa de un nuevo módulo de SOAR, el llamado *módulo de decisión* (Decision making), en el cuál se comparan los *operadores* almacenados en el paso anterior para decidir cuál de ellos será el seleccionado.
4. Si un *operador* se selecciona, la regla con la que enlaza ejecutará sus acciones en el cuarto paso del *ciclo de ejecución* ya que sus *condiciones*, que incluyen tener ese *operador* seleccionado, se cumplirán.

El *módulo de decisión* hace uso de los atributos de los *operadores* para seleccionar uno sobre los demás. El atributo principal para la decisión tiene unos valores concretos que puede tomar cada operador. Estos valores pueden ser:

- Best: Si un operador tiene este valor se selecciona automáticamente.
- Better: Cuando los *operadores* tienen atributos *acceptable*, el *operador* con un atributo **adicional** *better* será el seleccionado. Si existe más de uno se tendrán en cuenta otros factores entre los *operadores* que contengan este atributo.
- Acceptable: Como mínimo un *operador* debe tener un atributo *acceptable* para ser elegido. Si todos los *operadores* elegidos contienen únicamente atributos *acceptable* se generará un error (que SOAR es capaz de manejar) debido a que no se puede elegir y se utilizará un método especial para llegar a una conclusión.
- Indifferent: Este atributo se puede añadir de forma **adicional** cuando un *operador* posee un atributo *acceptable*. Si todos los *operadores* poseen el par de atributos *acceptable* e *indifferent* significa que todos son seleccionables pero no importa cuál sea el elegido, por lo que se hará de forma aleatoria. Esta aleatoriedad puede estar sesgada por algunos mecanismos de aprendizaje (ver punto 2.1.1).
- Worse: Si un *operador* tiene este atributo **adicional** junto con acceptable será el último en poder ser seleccionado. Si todos los *operadores* tuvieran este atributo se tendrían en cuenta otros factores.
- Reject: Este atributo prohíbe la selección del operador en cualquier caso. Si solo existe un *operador* seleccionable y tiene este atributo se producirá un error que SOAR deberá manejar.

Durante la ejecución el agente puede encontrarse con errores. La mayoría de estos errores se conocen y a veces se fuerza que sucedan, ya que ponen en marcha mecanismos que fomentan el aprendizaje. Estos errores se denominan *impasses (Impasse).* Los *impasses* que pueden suceder son los siguientes:

1- Estado sin cambios: este impasse surge cuando ningún *operador* es seleccionable. Ya sea debido a que tienen atributos *reject* o a que no se ha podido seleccionar ningún *operador* de la *memoria procedural*.
2- Nudo de operadores: este impasse surge cuando todos los operadores propuestos son indistinguibles para su elección, por ejemplo si todos poseen atributos únicamente *acceptable*.
3- Conflicto de operadores: este impasse surge cuando los operadores propuestos tienen conflictos de atributos better/worse que se contradicen entre sí, impidiendo su elección.
4- Operador sin cambios: este impasse surge cuando un *operador* continúa siendo seleccionado en sucesivas decisiones creando un bucle.

Cuando aparece un impasse, SOAR duplica su *estado* en la *memoria de trabajo* creando un espacio a parte llamado *sub-estado* donde se tratará de solucionar el problema. Cuando se llega a una solución todas las decisiones que se han tomado para resolver el impasse se aplican sobre el estado principal. El *sub-estado* es eliminado tras encontrar la solución.

2.1.1 Sistemas de aprendizaje

En SOAR existen varios tipos de aprendizaje. El aprendizaje a corto plazo (Chunking), el aprendizaje a largo plazo (aprendizaje por refuerzo) y el aprendizaje relacionado con las emociones (Appraisals) (ver Fig. 5).

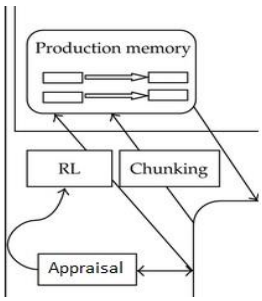

Figura 5 – Módulos de aprendizaje de la arquitectura SOAR

Chunking

El primer sistema de aprendizaje que ofrece la arquitectura SOAR es el Chunking[2]. Este sistema actúa cuando aparecen varios *operadores* que son indistinguibles a la hora de seleccionarse, por lo que hace uso de los *impasses* y crea *sub-estados* para poder llegar

[2] Desarrollo en el Anexo B: Técnica de aprendizaje: Chunking

a realizar una selección. Cuando el sistema hace la elección final del *operador* se produce un proceso de aprendizaje.

Cuando se produce un *impasse* (y se crea un *sub-estado* para resolverlo), la arquitectura entra en un proceso de **simulación**. Se denomina así debido a que el sistema crea el mismo número de *sub-estados* que de operadores involucrados en el *impasse* y aplica las *acciones* asociadas a cada *operador* en el *sub-estado* que le corresponde.

Una vez se han aplicado dichas acciones sobre los *sub-estados,* la arquitectura analiza cada uno de ellos y determina si se ha generado un nuevo estado que se acerque más a la *meta* que se pretende conseguir. El *operador* cuyas acciones asociadas han producido el mejor resultado para la resolución del problema será el seleccionado.

Tras esto, el sistema de aprendizaje analiza las **decisiones** implicadas en la mejor solución encontrada y crea una **nueva *regla*** *de decisión* que se añade a la *memoria procedural*. En ocasiones, en vez de crear nuevas reglas, las existentes se modifican. Esta forma de aprendizaje es **inmediata**, sólo requiere que se genere un *impasse* para crear una nueva regla.

Aprendizaje por Refuerzo

El segundo **sistema de aprendizaje** es el *aprendizaje por refuerzo* (Reinforcement Learning[3]**).** El agente aprende por **repetición** de la **experiencia** adquirida al encontrarse reiteradas veces en la misma situación y tener que tomar las mismas decisiones debido a la naturaleza de la tarea o del entorno.

Cuando la selección de *operador* se realiza entre *operadores* con atributos *indifferent,* se selecciona el operador de forma aleatoria. Aunque todos los *operadores* tienen la misma probabilidad de ser elegidos, el sistema de aprendizaje va dando o quitando peso a uno de los operadores frente a otros de forma que aquellos que dan mejores resultados tengan más probabilidad de seleccionarse.

Para que este sistema de aprendizaje funcione hacen falta algunas reglas especiales que otorgan recompensas positivas o negativas en función del *operador* seleccionado. Este sistema de aprendizaje es dinámico ya que puede adaptarse fácil y rápidamente a los cambios que se produzcan en el entorno. Esto se hace únicamente modificando las recompensas obtenidas en la elección.

Appraisals

El último de los tipos de aprendizaje es el **sistema de emociones** de SOAR (Appraisals**)**, que se utiliza para simular las emociones del agente. Este módulo está conectado a *la*

[3] Desarrollado en el Anexo C: Técnica de aprendizaje: Reinforcement Learning

memoria de trabajo y al sistema de *aprendizaje por refuerzo*. Las emociones en SOAR se tratan **individualmente** de forma matemática: se genera una ecuación que **retoca** el valor de los pesos que realiza por defecto *el sistema de aprendizaje*. Esto se lleva a cabo para que el agente se comporte deliberadamente de manera que parezca que la emoción le influye directamente.

2.1.2 Memorias a largo plazo

Además de la *memoria de trabajo* que se considera a **corto plazo** y *la memoria procedural*, imprescindible para el funcionamiento del sistema, existen otras dos memorias adicionales a **largo plazo**: la *memoria semántica* y la *memoria episódica*, que pueden activarse según las necesidades individuales del agente. Las memorias a **largo plazo** aparecen representadas a la derecha de la *memoria procedural* (ver Fig. 6).

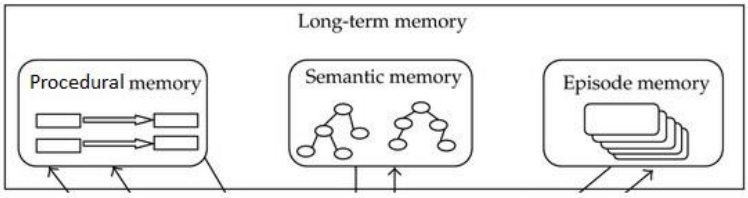

Figura 6 – Módulos de memoria de la arquitectura SOAR.

Memoria Semántica

A la primera de éstas memorias se le denomina *memoria semántica* (**Semantic memory[4])** y la información que se almacena en ella utiliza la misma codificación que la de la *memoria de trabajo*, basada en **estructuras simbólicas**.

Se define **concepto** como la representación mental de un objeto, hecho, cualidad, situación, etc… Cuando un conjunto de *entidades* con sus atributos aparecen de manera habitual juntas entonces SOAR lo caracteriza como un **concepto**.

A la *memoria semántica* también se le denomina memoria de **conceptos**, ya que los almacena cuando aparecen en la *memoria de trabajo* de forma que, aunque se modifique el *estado,* esa información siempre queda guardada para ser consultada en cualquier momento.

Por ejemplo: un agente con cierta experiencia en su entorno es capaz de darse cuenta de que el nodo "tabla" con 4 atributos "pata" es un concepto general de "mesa". El agente guarda este concepto para utilizarlo en el futuro si fuera necesario para ampliar su conocimiento.

Para acceder a esta memoria desde la *memoria de trabajo* se realizan consultas similares a las que se realizarían en un sistema SQL [12].

[4] Desarrollado en el Anexo D: Sistema de memoria: Semantic Memory

Esta memoria se puede utilizar de dos formas: con contenido cargado previo al inicio de la ejecución del agente, o con contenido que va creándose dinámicamente.

Memoria Episódica

La segunda memoria a **largo plazo** es la *memoria episódica* (Episodic memory[5]**)** y su representación interna también utiliza estructuras simbólicas, aunque estas estructuras son diferentes a las de la *memoria de trabajo* o a las de la *memoria semántica*, ya que la *memoria episódica* guarda una **"**foto**"** del **estado actual** en la *memoria de trabajo* con información del momento **(número de ciclo)** concreto en el que se encuentra.

De esta forma almacena un "álbum" de *estados* del sistema en una escala temporal, dándole información cuando se encuentre en un *impasse* para acceder a estados similares en los que ya se haya encontrado y al ver el resultado obtenido podrá acceder a la solución si esta ha sido buena, priorizando los "recuerdos" más recientes en sus entradas de memoria.

Para gestionar el espacio en esta memoria existen 2 formas distintas de almacenar la información, dependiendo de cómo queramos que funcione el sistema:

- Se puede almacenar la "foto" del *estado* cada vez que un *operador* se aplica, por lo que se tiene una traza perfecta de todos los cambios.
- Se puede guardar la información sólo cuando haya un cambio significativo que genere una salida hacia el entorno del agente. Esta segunda forma se utiliza para evitar almacenar una sucesión de estados similares al encontrarse el sistema en momentos de poco cambio, por ejemplo cuando el agente se desplaza de un sitio a otro físicamente.

2.2 Librerías de SOAR

Para la implementación de los programas propios de SOAR, más allá de la escritura de las reglas, se ha decidido utilizar el lenguaje de programación Java. Con él se modelará el comportamiento del entorno del agente.

Para conectar el sistema SOAR con Java existen dos APIs diferentes. La arquitectura SOAR de base está programada en C++ por sus creadores y tiene una API en este lenguaje. Sin embargo, debido al, cada vez más amplio, uso de Java, **Dave Ray** [7] creó una implementación completa de SOAR en Java, generando las librerías **JSoar** [6], las cuales permiten gestionar el kernel de SOAR y los agentes a ejecutar. Actualmente JSoar lo mantiene *Soar Technologies Inc. [5],* una empresa privada. Aun así la mayor parte del código JSoar es público.

[5] Desarrollado en el Anexo E: Sistema de memoria: Episodic Memory

Posteriormente los creadores originales de SOAR decidieron sacar también su propia librería, SML, que en este caso también conecta la arquitectura original, utilizando Java.

2.2.1 JSoar

Pros:

- Mejor encapsulamiento, totalmente enfocado a Java.
- Posibilidad de usar herramientas para la depuración.
- No necesita dependencias con las librerías en C++, evitando modificación de variables de entorno y otras configuraciones.
- Mayor cantidad de documentación para facilitar el comienzo de su uso.
- Facilidad de uso una vez se manejan listeners y handlers[6].
- Utilización paralela de un debugger, que facilita el proceso de aprendizaje de la propia programación de reglas, así como la depuración del programa.

Contras:

- Carencias debidas a la falta de actualización, ya que algunas de las posibilidades de SOAR no están implementadas aún en JSoar.
- El punto anterior enfatiza el problema de la simplicidad del sistema, útil para aprender y crear agentes sencillos, pero escaso en agentes para los cuales es necesaria la configuración de sus parámetros internos de SOAR.
- No es posible utilizar el sistema de generación combinatoria automática de reglas (sistema realmente útil para crear reglas automáticamente evitando tener que escribir a mano una a una con la única diferencia de un valor de una variable).

2.2.2 SML

Pros:

- Librerias/APIs totalmente actualizadas, ya que enlazan con el sistema original SOAR que se puede encontrar en la página web principal de la arquitectura, por lo que todas las posibilidades, que se sabe que ofrece SOAR, son programables.
- Posibilidad de crear clientes SOAR que se conecten a una máquina con ésta arquitectura, o un servidor que se encuentre en otro lugar.

[6] Desarrollado en Anexo G: Problemas encontrados y sus soluciones

- Todos los comandos de configuración de la arquitectura están disponibles, así como la utilización de la generación combinatoria de reglas.

Contras:

- Las librerías/Apis son una traducción a Java desde C++, por lo que su uso no es tan intuitivo.
- La información para la utilización de estas librerías (tutoriales o manuales) es muy escasa.
- Poca optimización del sistema. Pueden surgir fácilmente problemas con desbordamientos de pila que hay que tratar específicamente.
- La configuración del sistema para su uso es más compleja, teniendo que añadir variables de entorno y dependencias con las librerías originales.

En primera instancia se decidió seleccionar el sistema JSoar por su mejor adaptación a Java y su sencillez. Muchos de los agentes más sencillos que se han realizado durante el proceso de aprendizaje se crearon con estas librerías, sin embargo se llegó a un punto del proyecto en el cuál era necesario modificar los valores internos de la ecuación de *aprendizaje por refuerzo* con objeto de poder modificar el comportamiento de aprendizaje en el agente, lo cual era fundamental para investigar y aprender a usar este método.

Debido a la imposibilidad de realizar esa tarea con JSoar se pasó a utilizar el sistema SML de forma que pudieran hacerse los cambios de los valores internos y se observó que, una vez se aprende a crear agentes y programas SOAR en JSoar, utilizar SML es relativamente sencillo y provee muchas más posibilidades y un conocimiento más detallado de la arquitectura. El uso de SML es totalmente recomendable con agentes complejos.

De esta manera el agente presentado en este proyecto se ha programado utilizando las librerías SML.

3. Problema modelado

Esta sección se divide en 3 bloques:

- El **primer bloque** ofrece una visión general del problema, definido como la relación de un agente en un entorno determinado (que se va a denominar *mundo* (punto 3.1)) y se presenta la información del problema modelado, junto con sus características (punto 3.2).
- El **segundo bloque** profundiza en la integración del problema en la arquitectura SOAR (punto 3.3) a través de cuatro secciones que destacan los módulos más importantes utilizados para llevar a cabo la implementación.
- El **tercer bloque** presenta la implementación en Java del *mundo* y la comunicación con el agente.

3.1 Relación entre el agente y el mundo

Cuando se crea un agente con la arquitectura SOAR se deben tener en cuenta tanto las reglas que conformarán el comportamiento del agente como su relación con el entorno (*mundo exterior*).

En este caso, el mundo se simula virtualmente utilizando el lenguaje de programación Java (ver Fig. 7). Se sustituyen los posibles sensores, de los que la arquitectura pudiera conseguir la información de su entorno, por "paquetes" de información generados en el entorno Java. Las órdenes producidas por el agente que interaccionan con el entorno (como la acción "moverse a la derecha") se tratan para que se vean reflejadas en la interfaz. De esta manera se puede realizar un seguimiento de las decisiones tomadas por el agente de forma visual.

Además de **crear el mundo exterior** del agente, el entorno Java proporciona un **sistema de control y de tiempo** que se gestiona a través de una interfaz gráfica. Es importante destacar que, como en cualquier entorno real, pueden suceder imprevistos, y estos también se han contemplado y se generan de forma algorítmica.

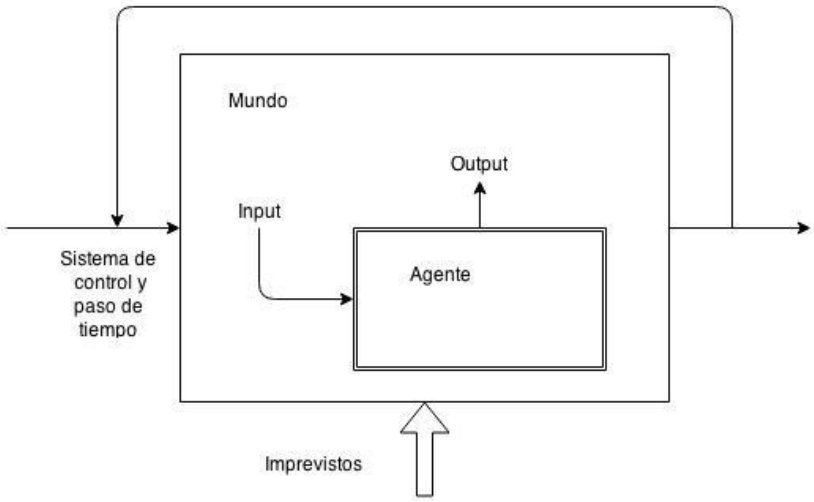

Figura 7 – Diagrama de relación agente-mundo

3.2 Descripción del problema

El agente hace uso de un conjunto de capacidades cognitivas que le son provistas inicialmente para poder resolver los problemas que se le presentan. Además es capaz de interactuar con su entorno para cubrir una serie de necesidades que le permiten sobrevivir. El objetivo final es sobrevivir el máximo tiempo posible.

Las necesidades que tiene el agente son: *hambre, sed* y *seguridad*. Se presupone un **entorno hostil**, por lo que la necesidad *seguridad* no es trivial y debe priorizarse cuando el hambre o la sed no supongan un riesgo inmediato. En el mundo existe una zona de seguridad y el agente deberá **maximizar el tiempo de estancia en la zona segura** con objeto de aumentar sus posibilidades de supervivencia.

En un entorno real pueden surgir imprevistos para los cuales no siempre se puede reaccionar a tiempo. El agente puede morir de forma súbita debido a su entorno. Por ejemplo, la comida que sacia su necesidad de hambre puede estar **envenenada** y el agente no es capaz de saberlo previamente, por lo que muere al consumirla. Otros imprevistos ante los que sí se puede reaccionar serán los cambios en ciertos comportamientos del entorno que serán expuestos más adelante.

El *mundo* completo en el cual se encuentra el agente consta de: dos pozos de agua, un refugio y una zona de alimentación. Por simplicidad se han dispuestos de tal forma que el refugio quede en el centro entre los pozos y a la misma distancia de la zona de alimentación (ver Fig. 8).

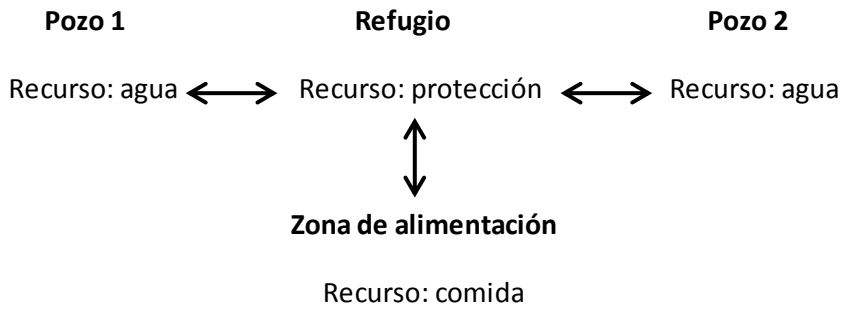

Figura 8 – *Diagrama de zonas del mundo*

Los pozos no tienen por qué disponer de *agua* en todo momento. El agente sólo podrá consumir el recurso *agua* de un pozo cuando éste se encuentre disponible. Así mismo, en la zona de alimentación, el recurso *comida* variará sus características a lo largo del tiempo.

El agente no posee una **visión global** del mundo en el que se encuentra. Cuando se sitúa en el refugio no puede determinar de forma trivial si un pozo está lleno o vacío, o qué características posee la comida que se encuentra en la zona de alimentación. Sólo es consciente de lo que hay en la zona concreta donde se encuentra en cada momento y de sus necesidades vitales, por lo que debe desplazarse de una zona a otra para conseguir información u obtenerla haciendo uso de sus capacidades cognitivas.

Las posibles acciones que puede realizar el agente varían entre las zonas: si el agente se encuentra en un pozo, además de la posibilidad de desplazarse, tiene la opción de consumir el agua si está disponible. Los pozos tienen un patrón de disponibilidad de agua entre uno u otro que el agente debe aprender. En ocasiones sólo habrá agua en uno de los pozos pero, por defecto, al inicio de la simulación del entorno el patrón de aparición es el siguiente: cuando el agente consume agua de uno de los pozos, dicho pozo se vacía y se llena el contrario.

Las acciones que puede realizar en el refugio comprenden el desplazamiento a otra zona o "consumir el recurso de seguridad" que equivale a quedarse quieto en el refugio, donde las posibilidades de morir se reducen a morir de sed o de hambre si alguna de las dos necesidades alcanza su valor máximo y no es cubierta.

En el mundo que se modela existen cinco tipos diferentes de comidas, de las cuales se presentan dos cada vez que el agente accede a la *zona de alimentación*. La comida disponible va cambiando creando distintas combinaciones.

Cuando el agente toma la decisión de comer, debe en primer lugar elegir la cantidad de comida que va a consumir en función del valor que en ese momento posee de *hambre*. La cantidad de comida que saciará su hambre se determina en calorías.

Los tipos de comida con los que se puede encontrar el agente poseen un atributo que representa su valor calórico, por lo que el agente deberá seleccionar la cantidad de piezas de cada tipo que debe consumir para cubrir su necesidad.

Además, debe utilizar sus capacidades de aprendizaje para agilizar esta tarea en sucesivas situaciones similares.

Durante el tiempo que dure la ejecución, el agente puede morir de *sed*, *hambre* o de *un imprevisto* del entorno, finalizando su funcionamiento.

3.3 Integración del problema en SOAR

El comportamiento del agente, la toma de decisiones y el intercambio de información con el entorno se codifican mediante *reglas de decisión*, las cuales se cargan en la *memoria procedural* ofrecida por la arquitectura SOAR y se utilizan para llevar a cabo los *ciclos de ejecución*.

El agente que responde al problema presentado en el punto 3.2 se implementa utilizando 97 *reglas de decisión* escritas a mano[7] y 300 adicionales generadas por el sistema para el correcto funcionamiento del *aprendizaje por refuerzo*. Las 97 *reglas* se pueden agrupar por objetivos o similitud (ver **Tabla 1**).

Tabla 1 - Reparto de reglas

2	Reglas de inicialización.
9	Reglas de gestión de la muerte por el incumplimiento de las necesidades.
2	Reglas de gestión de la memoria episódica.
18	Reglas de gestión del movimiento del agente.
9	Reglas de gestión para consumir los diferentes recursos disponibles.
8	Reglas de gestión de la memoria semántica.
27	Reglas para la elección de la comida a consumir.
7	Reglas de gestión de los sistemas de aprendizaje Chunking y aprendizaje por refuerzo.
12	Reglas de gestión de las necesidades y control del aprendizaje.
3	Reglas de monitorización y gestión de la salida hacia el entorno Java.

Estas reglas hacen uso de algunos módulos de memoria y aprendizaje de SOAR para proporcionar diversas capacidades cognitivas al agente con el fin de que pueda alcanzar su *meta* de supervivencia. Las siguientes sub secciones detallan estas relaciones.

[7] Las reglas pueden consultarse en el Anexo F: Reglas de decisión del agente modelado.

3.3.1 Utilización de aprendizaje por refuerzo

El **sistema de aprendizaje por refuerzo** es el hilo conductor de la solución del problema. El *aprendizaje a largo plazo por recompensas* dota al agente de la capacidad de aprender a tomar las decisiones adecuadas para adaptarse a su entorno maximizando sus probabilidades de sobrevivir y, al mismo tiempo, modifica el comportamiento del agente cuando se produce algún cambio en el entorno.

Al codificar reglas que generan recompensas positivas cuando el agente realiza una acción beneficiosa, se van modificando los valores de probabilidad de elección de los *operadores* que la han generado. Ciclo a ciclo los porcentajes de elección permiten que casi siempre se tome la mejor decisión.

El *aprendizaje por refuerzo* influye en toda la gestión de desplazamientos del agente, recompensa la decisión de movimiento cuando una necesidad alcanza valores elevados, y penaliza en el caso contrario. Este aprendizaje afecta de la misma manera a la decisión de consumir o no un recurso disponible. Sin embargo, en la zona de alimentación, el hecho de consumir el recurso *comida* provoca un aprendizaje por refuerzo demasiado lento, y por este motivo se ha decidido utilizar otro sistema de aprendizaje para ésta situación en particular.

3.3.2 Incorporación de la memoria episódica

El *sistema de aprendizaje por refuerzo* funciona de forma óptima para la toma de decisiones de acciones sencillas, como consumir el recurso *agua* si el agente se encuentra en un pozo donde el *agua* está disponible y su nivel de *sed* es alto. Sin embargo, cuando se pretende que el agente sea capaz de adaptarse al patrón de aparición de agua de los pozos (por defecto de forma alternativa entre ellos) es necesario incorporar la utilización del **módulo de memoria episódica**.

Éste módulo, que ofrece la posibilidad de acceder a los recuerdos de los *estados* y las decisiones tomadas en una escala de tiempo, es la pieza clave del *aprendizaje de patrones* por parte del agente. Cuando el agente consume el recurso *agua* por primera vez en un pozo, ésta acción se almacena de forma permanente en la memoria episódica y, en la siguiente situación en que el nivel de *sed* del agente sea alto, podrá consultar esta memoria para determinar de qué pozo bebió la última vez.

Teniendo acceso a esta información y tras unas pocas decisiones acertadas o erróneas, el *sistema de aprendizaje por refuerzo* converge en el siguiente comportamiento: cuando el nivel de sed del agente es alto, la probabilidad más elevada es que se decida realizar la acción de **consultar en la memoria episódica** cual fue el último lugar en el que se consumió *agua*. Tras esto, el agente decide desplazarse al pozo contrario y consume el *agua* de ese pozo. Este orden de decisiones conforma el comportamiento óptimo para el agente.

3.3.3 Contenido de la memoria semántica

En el problema presentado se precisa conocer de antemano las **características de las piezas de comida** disponibles en el entorno. Entre estas características se encuentra el **contenido calórico**, necesario para poder realizar la elección en la *zona de alimentación,* y así poder cubrir la necesidad *hambre*.

Como cada comida posee características propias (concibiéndose de esta manera como *conceptos* individuales), la *memoria semántica* ofrece el **sistema de almacenamiento de información** óptimo para éstos datos.

La carga de los datos se realiza previa al comienzo de la ejecución del agente, de forma que tenga acceso a los datos desde el principio. Cuando éste se encuentra en la *zona de alimentación,* adquiere la información "visual" de la comida que puede elegir. Esta información consiste en un atributo "nombre" de la pieza de comida en concreto. Sin embargo, debido a que precisa una información más completa, el agente realiza una consulta a su memoria semántica utilizando ese atributo "nombre" y completa su conocimiento descargando de la memoria el resto de la información.

Debido a que el tiempo de consulta a la memoria semántica es muy bajo, la eficiencia de este método le hace similar al funcionamiento de una memoria biológica, sin el riesgo de pérdida de información.

3.3.4 Chunking

Una vez el *agente* sabe la cantidad de calorías que debe consumir para saciar su *hambre* debe proceder a **seleccionar** las piezas de comida que tengan, en total, esa misma cantidad de calorías. Este problema de seleccionar las piezas de comida necesarias se va a convertir en un sub objetivo, **una sub meta**, que el *agente* debe alcanzar antes de proseguir con su ejecución.

Cuando esta **sub meta** se haya alcanzado, habiendo decidido qué piezas escoger para saciar el nivel de *hambre* que tiene el *agente*, el proceso de aprendizaje *Chunking* convertirá ésta decisión en una *regla* nueva. Por tanto la próxima vez que el *agente* se encuentre en la misma situación sabrá que piezas de comida seleccionar de forma inmediata.

Se va a utilizar un ejemplo para explicar el uso del Chunking en el problema:

El agente se encuentra en la *zona de alimentación* frente a dos tipos de comida: "A" y "B". Su nivel de *hambre* indica que debe elegir una cantidad de comida equivalente a 600 calorías para cubrir su necesidad. Como ya dispone de toda la información de las piezas de comida, sabe que "A" posee un valor de 200 calorías y que "B" posee un valor de 500. Su objetivo es conseguir reunir las 600 calorías y sus posibilidades son:

- Coger una o varias piezas de comida.
- Si ya posee una pieza de comida puede partirla por la mitad, de forma que sólo consumiría media.
- O deshacerse de una pieza entera.

Por lo que se observa que, con solamente una de estas decisiones, no se podría cumplir el objetivo de consumir 600 calorías. El sistema crea un *impasse*, generando un *sub estado* en el cuál se van a simular muchas posibles sucesiones de decisiones hasta que una de ellas concluya en la consecución de la **sub meta**. En este caso, por ejemplo, una solución válida podría ser coger una pieza de "B" y media pieza de "A".

Ya que los *operadores* propuestos para solucionar esta **sub meta** (coger A, partir A, dejar A…) tienen todos sólo el atributo "acceptable" no se sabe cuál es mejor elegir a priori. Se crea un impasse, gracias al cual se pone en marcha el sistema de aprendizaje ***Chunking***, que tomará una decisión válida en cada caso y guardará la información en forma de *regla de decisión.*

3.4 Integración del problema en Java

La arquitectura SOAR, dentro de la cual se ejecuta el agente, utiliza un hilo de ejecución (Thread) individual; por lo tanto el programa en Java se ejecutará en un hilo diferente. Se han desarrollado los métodos de comunicación entre ambos hilos.

El código Java que ha sido desarrollado para este problema cubre diferentes aspectos:

- En primer lugar, se encarga de modelar el *mundo* en el que se mueve el agente y con el cual interactúa. Debido a que el mundo es un entorno virtual, el código Java se encarga de simular todos los parámetros que el agente precisa para esta interacción.
- En segundo lugar, se encarga de gestionar el paso de información de entrada y de salida con el agente, ejecutado en un hilo diferente.
- En tercer lugar, crea la interfaz gráfica donde se pueden observar los resultados derivados de la ejecución de cada ciclo del agente.

En esta sección se presentan tanto el aspecto primero como el segundo, ya que el tercero forma parte de la sección de resultados (punto 4).

La clase principal del código Java, donde se encuentran los procedimientos que hacen de API de conexión con SOAR, consta de 15 funciones, 6 de ellas corresponden con la conexión con SOAR. Éstas son:

- La **inicialización del hilo** que ejecuta el simulador de la arquitectura. En el simulador se cargan los documentos que contienen las reglas del agente.
- La **creación de los "listeners"** que permiten las interrupciones del sistema SOAR para interactuar entre ambos hilos de ejecución.

- Las funciones de **gestión de entrada-salida**, donde se modifica el estado global del agente, se guarda la información para mostrar sus decisiones y se le pasa la información del entorno.
- Dos **funciones de utilidad**, de las cuales una se utiliza para finalizar el hilo de SOAR cerrando el simulador desde Java en vez de esperar a que "muera" el agente.
- Y una última función que ejecuta **comandos** en el simulador para cambiar configuraciones, o mostrar los datos que se le pidan.

Los tiempos medios obtenidos de 20 ejecuciones diferentes del agente son los siguientes (ver Tabla 2).

Tabla 2 - Tiempos de ejecución en ms	
Tiempo de preparación del agente (Java)	2
Tiempo de inicialización del agente (SOAR)	33
Tiempo medio de ciclo (SOAR)	6
Tiempo medio de gestión de Entrada-salida (JAVA)	1

Tiempo medio de la acción "consumir comida"	
Por primera vez	500
Tras aprendizaje	24

Para crear una ilusión de "*paso del tiempo*" del agente se utilizan interrupciones de la ejecución de Java en función del tiempo que se ha considerado que es necesario que tarde en realizar cada tarea.

Se ha decidido una conversión temporal de forma que cada 5 minutos estimados de la vida del agente se correspondan con un segundo real, aunque la toma de decisiones de SOAR tarda centésimas de segundo en ejecutarse.

3.4.1 Listeners

Debido a que la implementación consta de dos hilos de ejecución diferentes, el de la arquitectura SOAR y el de JAVA, el paso de información de uno a otro no se realiza de manera trivial. Se ha decidido utilizar un sistema de Listeners que ofrecen las librerías para Java de SOAR. Este sistema crea interrupciones desde el hilo de SOAR hacia el hilo de Java para indicarle diferentes informaciones, por lo que se debe preparar el hilo de Java para capturar éstas interrupciones y saber interpretarlas.

Se han elegido como ejemplo dos pares de interrupción - listener (que se desarrollan a continuación): el paso de mensajes y el fin de ciclo de ejecución del sistema SOAR.

Cada vez que en la arquitectura SOAR se genera una salida en forma de texto (asociada a una decisión) se crea una interrupción a Java del tipo de paso de mensaje. De esta forma al capturar esta interrupción se puede escribir la salida en la consola de Java o en la interfaz gráfica que se utilice. De esta manera es posible seguir la toma de decisiones tal y como las va tomando el agente.

De la misma manera cada vez que se acaba un ciclo de ejecución del agente la arquitectura SOAR genera una interrupción hacia el hilo de Java, ya que entre el fin de un ciclo y el comienzo del siguiente es el momento de gestionar las salidas proporcionadas por el agente hacia el entorno y crear las entradas que se le pasarán al nuevo ciclo de ejecución que está por comenzar.

En este punto el hilo de SOAR detiene su ejecución para permitir a Java ponerse al día y crear su nueva información antes de permitirle al hilo continuar.

3.4.2 Datos de entrada y salida

El hilo de ejecución de Java considera los datos de salida del agente SOAR como sus entradas, y los datos de entrada del agente como sus salidas (ver Fig. 9).

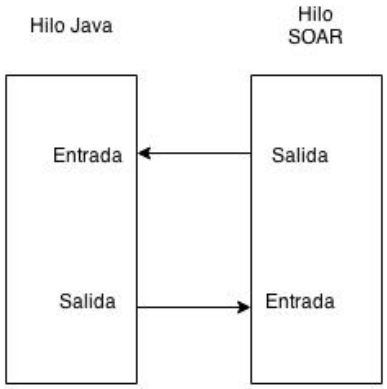

Figura 9 – Gestión de entrada y salida entre Hilos.

Estos datos de entrada en el hilo de Java son los que proporciona el agente al terminar un ciclo de ejecución. Contienen diversas acciones que modifican la posición del agente en su entorno o información sobre sus necesidades vitales.

Nada más terminar el *ciclo de ejecución* en el *agente* se procesa esta información de salida en el hilo de Java y se combina con la que ya se tenía para generar los datos de salida, como datos de entrada para el hilo de SOAR.

Si la acción realizada por el agente ha sido un desplazamiento, se enviará desde el hilo de Java al agente la información de su nuevo emplazamiento en la zona a la cual se haya movido. Si la acción ha sido consumir un recuso se procederá a procesar si éste debe dejar de estar disponible o no en el futuro.

Como se ha mostrado antes el agente SOAR utiliza una codificación simbólica para representar toda su información, sin embargo Java no puede interpretarla, ni enviar sus tipos de datos al agente sin traducirlos previamente. Se han creado dos funciones que gestionan esta conversión.

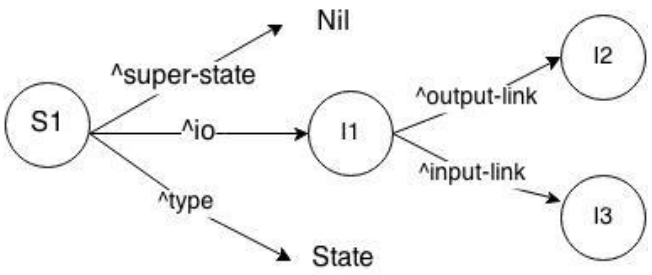

Figura 4 Bis – Estado básico de un agente

EL agente SOAR tiene por defecto dos entidades (ver Figura 4 bis), "I2" e "I3" a las cuales tiene acceso el hilo de Java para: leer datos en el caso de I2 (enlace de salida) y escribir los datos de entrada (enlace de entrada).

4. Resultados

Se ha decidido presentar los resultados del problema modelado a través de una interfaz gráfica generada en Java.

Estos resultados muestran el comportamiento del agente y la sucesión de decisiones que va tomando. Estas decisiones se visualizan mediante una representación gráfica del problema.

Se muestran además los valores de las necesidades que tiene que cubrir en cada momento el agente y la decisión que se toma en el instante en que consume comida en la *zona de alimentación*.

4.1 Interfaz gráfica

La interfaz se compone de una ventana con una serie de botones, zonas en las que se muestran resultados y una entrada de comandos (ver Fig. 10).

Figura 10 – Ventana de la interfaz.

La parte de la interfaz más importante para poder sacar conclusiones, es el tiempo relativo que pasa desde que el agente tiene un comportamiento errático hasta que se puede vislumbrar que crea ciertos patrones. El tiempo se representa mediante un cursor que va moviéndose de izquierda a derecha sobre una línea numerada que indica cuantas horas relativas han pasado para el agente (ver Fig. 11).

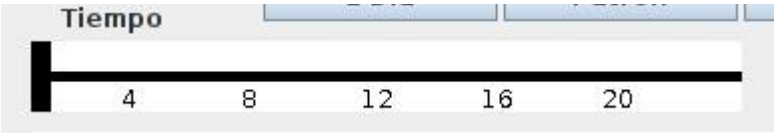

Figura 11 – Barra de tiempo de la interfaz.

La interfaz ofrece diferentes botones (ver Fig. 12):

- Iniciar y finalizar la ejecución del agente (“Iniciar” y “Salir”), y mostrar la ejecución de un día de la vida del agente (“1 día”).

- Por otro lado alterar configuraciones o crear imprevistos, como modificar el tipo de comida ofrecida en la *zona de alimentación*, cambiar los patrones de aparición de agua, o convertir parte de la comida en veneno ("Comida", "Patrón" y "Veneno").
- También existe la opción de ver paso a paso las decisiones del agente en tiempo real de cómputo, pudiendo hacer las modificaciones sin tener que esperar a que acabe la "simulación" de un día ("1 Decisión").
- Por último se ofrece la posibilidad de reiniciar desde la interfaz la ejecución del agente si éste muere ("Reiniciar").

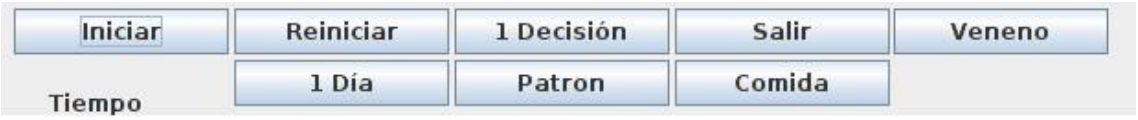

Figura 12 – Botones de la interfaz.

Puede observarse en todo momento la información de los valores de las necesidades vitales del agente: sed y hambre (la necesidad de seguridad es continua).

Se muestra en diferentes líneas los resultados de la decisión tomada en la *zona de alimentación* (ver Fig. 13).

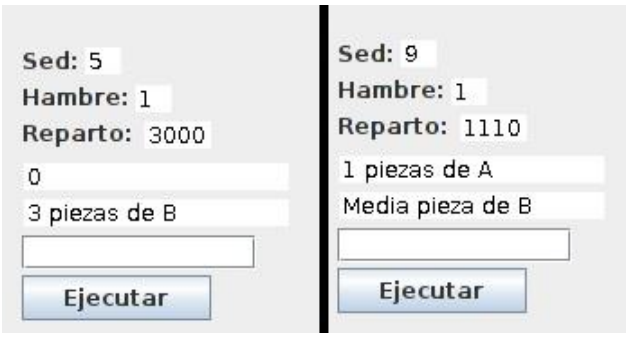

Figura 13 – Dos ejemplos de las necesidades y decisiones sobre comida tomadas por dos agentes.

Existe un campo de texto en el que se pueden introducir posibles comandos que modifiquen la configuración interna de la arquitectura o bien para pedir que se muestre parte del contenido de las memorias de trabajo o a largo plazo (aunque el resultado aparece en la consola de Java ya que se espera que se use solo para pruebas y verificación).

Por último existe una zona más visual creada usando pequeñas imágenes denominadas clips [11], donde se muestra la actividad del agente según va ocurriendo. Estos gráficos muestran las decisiones tomadas por el agente (ver Fig. 14).

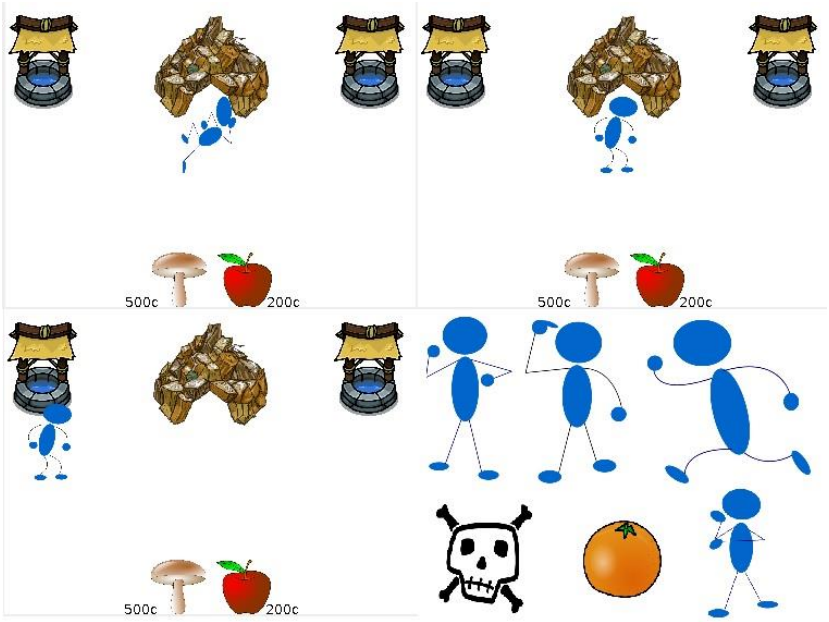

Figura 14 – gráficos del problema y otros clips utilizados

4.2 Resultados del comportamiento

El *agente* nunca alcanza un punto en el que pueda decirse que su comportamiento sea óptimo ya que, pese al aprendizaje, el mundo en el que se mueve es un mundo aleatorio, cambiante.

Sin embargo en base a las pruebas realizadas, se ha llegado a la conclusión de que necesita aproximadamente 250 *ciclos de ejecución* para "aprender" de forma óptima el patrón de aparición de agua de los pozos y la decisión de comer. En esta cuenta se excluyen los *ciclos* que se ejecutan en los *sub-estados*, por ejemplo cuando se decide qué comida va a tomar el *agente*.

El número de *ciclos* necesarios para el proceso de aprendizaje en estos *sub-estados* que se generan al entrar en la *zona de alimentación* varía en cada ejecución del *agente*. Esto ocurre porque a veces llega a la solución rápidamente (10 o 12 *ciclos*) y otras puede demorarse varios segundos de tiempo real (250 o 300 *ciclos*) debido a la aleatoriedad que existe.

Sin embargo, cuando ya ha aprendido una combinación concreta (necesidad y tipos de comida), el número de *ciclos* que tarda en consumir el recurso comida es siempre similar y no suele exceder los 10 *ciclos* SOAR.

5. Diagramas de tiempo

6. Conclusiones y trabajo futuro

Para este proyecto se ha estudiado e investigado la arquitectura cognitiva SOAR. Se han utilizado estos conocimientos para crear un agente que hace uso de los sistemas principales de SOAR y evoluciona con el paso del tiempo, adaptándose a los cambios de su entorno. Este agente utiliza en cada momento todo el conocimiento del que dispone para maximizar sus posibilidades de supervivencia. El agente cubre sus necesidades de *hambre* y *sed* mientras aprende a mantenerse el mayor tiempo posible resguardado, preservando su *seguridad.* Tal y como se propuso en el objetivo del proyecto fin de carrera.

Además el agente interacciona con el *mundo* cambiante a través de un programa Java que hace uso de las librerías de SOAR "SML", que han sido seleccionadas como la API de comunicación entre Java y la arquitectura SOAR. El programa en Java modela el *mundo* en el que se mueve el agente y gestiona los imprevistos que puedan aparecer.

Se ha creado una interfaz para tener cierto control sobre el agente que permite:

- Iniciar la ejecución.
- Ejecutar sólo un ciclo o el equivalente a un día en la vida del agente.
- Gestionar algunos de los imprevistos que pueden aparecer en el entorno.

Esta interfaz muestra los valores de las necesidades del agente y sus decisiones en el tiempo.

Líneas de trabajo futuro

Las dos líneas de trabajo futuro en torno a este proyecto son:

- La implementación del agente en un sistema real, en el cual habría que modificar las salidas del sistema para adecuarlos a un robot y hacer una gestión de las entradas basada, probablemente en visión espacial.
- Crear varios agentes simulados en un mismo entorno simultáneamente, con las mismas características, de forma que sean capaces de "interactuar" aprendiendo de los errores o aciertos de los demás, en vista a utilizarlos como simulación de una población y su adaptación al entorno.

Valoración personal

El haber trabajado con un sistema tan diferente a todos los que había estudiado a lo largo de la carrera me ha ofrecido la oportunidad de tener que aprender a pensar en la programación de una forma nueva y distinta, enriqueciendo así mi formación previa.

Añadir que, aunque no es un sistema totalmente óptimo, es potente desde el comienzo del aprendizaje de sus módulos y podría ser aplicado a multitud de campos y proyectos de forma satisfactoria dotándolos de nuevas posibilidades.

El único punto negativo de la experiencia de trabajo es la poca cantidad de información disponible, tanto de documentación teórica como de códigos de ejemplo y sistemas de ayuda personalizada. Es un sistema creado por unas pocas personas de la universidad del MIT y no tiene una infraestructura creada con todos estos servicios.

Sí que es posible encontrar ejemplos para problemas sencillos de lógica, como puzles o juegos, sin embargo no lo es para agentes complejos que simulen capacidades de más alto nivel.

Bibliografía

[1] John E. Laird; The Soar Cognitive Architecture. The MIT PRESS; 2012
ISBN 978-0-262-12296-2.

[2] Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: an architecture for general intelligence. Artificial Intelligence, 33(1):1-64. (document), 4.1.1, 4.1

[3] Lehman, J. F., Laird, J. E., Rosenbloom, P.; A gentle introduction to Soar, architecture for Human Cognition: 2006 update.

[4] Definición e información general:
http://en.wikipedia.org/wiki/Soar_(cognitive_architecture)

[5] Página principal:
http://soartech.github.io/jsoar/
http://www.soartech.com/

[6] Acceso al código de las clases de las librerías JSoar:
https://code.google.com/p/jsoar/source/browse/jsoar-core/src/main/java/org/jsoar/kernel/
http://darevay.com/jsoar/current/docs/jsoar-core/api/

[7] Guias y tutoriales de uso de JSoar por Dave Ray:
https://code.google.com/p/jsoar/w/list

[8] Página principal:
http://soar.eecs.umich.edu/

[9] Acceso al código de las clases de las librerías SML:
http://winter.eecs.umich.edu/~jzxu/sml/sml__ClientKernel_8h_source.html

[10] Guías básicas de SML: How to compile SML Clients, Quick start guide, Threads in SML y Output link guide:
http://soar.eecs.umich.edu/articles/articles/soar-markup-language-sml

[11] Imágenes png utilizadas en la interfaz:
http://www.clker.com/

[12] Date, C.J.; Darwen, H. (1997). A guide to the SQL Standard (4.a ed.). Reading, Massachusetts: Addison-Wesley.