Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Inteligencia Artificial

Módulo de Planeación en Robots de Servicio

Tesis

Que para optar por el grado de

Maestro en Ciencias de la Computación

Presenta

Ing. Ricardo Adolfo Fierro Villaneda

Director de Tesis:

Dr. Luis Alberto Pineda Cortés
Instituto de investigaciones en Matemáticas Aplicadas y Sistemas

Ciudad Universitaria, Cd. Mx. Mayo 2017

Índice

Lista de figuras

Lista de códigos

1 Introducción

Un robot de servicio es una entidad mecánica que ayuda a los seres humanos en diferentes tipos de tareas de forma autónoma, por lo general para la realización de trabajos peligrosos o repetitivos. Actualmente existen diferentes tipos de robots de servicio, los cuales están programados para cumplir tareas no específicas, pero sí dentro de un contexto concreto. A diferencia de los robots de manufactura -los cuales son eficientes pero repetitivos y no les afecta su entorno ya que sólo sirven para una función específica- los robots de servicio están pensados para realizar diferentes funciones. Por ejemplo, un robot de rescate cambia de ambiente en cada operación, pues debe ubicarse en el lugar de una catástrofe.

Para funcionar apropiadamente, un robot de servicio necesita ser capaz de observar su entorno (por medio de cámaras u otro tipo de sensores) y manipularlo. Usualmente esta manipulación se hace por medio de brazos metálicos. También es posible que deba recibir estímulos de usuarios en caso de encontrarse en un ambiente donde se requiera de interacción humano-robot. Debido a que las tareas a realizar pueden cambiar en cada ocasión, tomarán decisiones considerando su ambiente para funcionar autónomamente.

Aunque se busca desarrollar robots que puedan realizar todo tipo de tareas por medio del aprendizaje, por ahora la mayoría solo trabajan en áreas específicas.

El robot de servicio con el cual se trabaja en esta tesis es Golem-III, desarrollado por investigadores del Departamento de Ciencias de la Computación del Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas (IIMAS). Está pensado para interactuar con diferentes usuarios y, por lo tanto, sería útil que la forma en que tome sus decisiones se asemeje más a la forma en que lo hace un humano.

1.1 Planteamiento del problema

Actualmente, Golem-III utiliza un módulo de inferencia para solucionar problemas o resolver errores en escenarios donde toma el papel de un mesero o empleado de una tienda, el cual se divide en tres partes:

- Módulo de diagnóstico, el cual determina la causa de un error detectado en el escenario. Toma como parámetro el mundo observado hasta el momento.

- Módulo de decisión, el cual crea una decisión para solucionar un error encontrado. Toma como parámetros el diagnóstico generado por el módulo de diagnóstico, el mundo como se conoce actualmente y el mundo al que se desea llegar, según sus preferencias.
- Módulo de planeación, el cual determina los pasos a seguir para concretar dicha decisión. Toma como parámetros la respuesta del módulo de toma de decisión y el mundo como se conoce.

El problema que se presenta para Golem-III es que debe tomar el papel de un empleado en una tienda y esto implica poder interactuar con empleados y clientes. Por parte de los empleados, Golem-III puede recibir indicaciones sobre los productos y hacer suposiciones sobre sus acciones, y en el caso de los clientes recibir órdenes.

El escenario que usaremos para hacer las pruebas se diseñó para demostrar en la competencia *RoboCup* las capacidades de Golem-III. Un empleado y un cliente le dan algunas indicaciones a Golem-III, y éste debe ser capaz de seguirlas, corregir los errores que se presentan y satisfacer al cliente.

1.2 Objetivos

El objetivo es desarrollar un módulo de inferencia que contenga tres sub-módulos mencionados en la sección anterior. Este trabajo se concentra en analizar un módulo de planeación en el lenguaje Answer Set Programming (ASP) en el escenario ya desarrollado y hacer el análisis de la eficiencia del mismo, sus limitaciones y compararlo con el que tiene actualmente Golem-III.

Además de ver la capacidad de ASP para la modelación de tareas de inferencia, podremos hacer la comparación de la implementación de estas tareas, para ver cómo se diferencia la forma y el resultado de ASP (el cual se hace escribiendo las reglas de forma declarativa) con el del módulo de inferencia actual, el cual requiere que explícitamente se programe una heurística para búsqueda de soluciones.

Con todo esto, se pretende evaluar si es posible y conveniente utilizar Answer Set Programming para resolver problemas de inferencia en robots de servicio.

Además de esto, al desarrollar y comparar el módulo de inferencia, estamos creando un modelo de la actividad que el humano realiza de forma intuitiva, pues se sigue una lógica

similar a la que se implementa en Golem: Buscar una explicación de lo que pasó, decidir qué se puede hacer al respecto y desarrollar un plan para llevarlo a cabo.

1.3 Justificación

Las razones por las que trabajaremos con Answer Set Programming son las siguientes: El lenguaje Answer Set Programming ha sido utilizado en varias áreas diferentes, incluyendo inteligencia artificial y ha resultado útil para resolver problemas computacionalmente complejos. Además -por la forma en que trabaja- permite modelar las respuestas de una manera más natural de como lo hace el módulo de inferencia actualmente, así como modelar y analizar varios conjuntos de respuestas. Por estas razones, se cree que este lenguaje puede ser útil para el análisis y organización de la información en un robot de servicio.

1.4 Hipótesis

Debido a que Answer Set Programming está pensado para resolver problemas de alto costo computacional y que ya se ha utilizado para resolver diferentes problemas en áreas de ciencia y tecnología, la hipótesis es que es posible modelar planeación para Golem-III en el escenario donde corre actualmente. Sin embargo, es necesario analizar las capacidades y limitaciones de este lenguaje para ver qué tan viable es utilizarlo en este escenario y en futuros trabajos.

Para generalizar el uso del módulo de planeación, se crearán un conjunto de reglas específicas que tomen en cuenta las diferentes acciones que Golem-III puede realizar actualmente. Esto se hará con el objetivo de que no esté limitado por alguna acción o evento del escenario y se pueda expandir a cualquier tipo de situación dentro de su área de trabajo.

1.5 Organización de la tesis

En el capítulo 2 se aborda el robot Golem, su arquitectura y la forma en que trabajan los modelos de diálogo (con el cual realiza diferentes tareas). Además se habla sobre planeación en inteligencia artificial y se mencionan las características de Answer Set Programming y qué lo hace diferente de otros lenguajes como Prolog (el cual es el lenguaje que utiliza Golem actualmente para el módulo de inferencia), así como el sistema que se utiliza para correrlo, llamado DLV.

El capítulo 3 habla sobre el módulo de inferencia que se utiliza para resolver los diferentes problemas que Golem puede presentar dentro del escenario establecido en una tienda (por ejemplo, que los productos no se encuentren en su lugar correspondiente), así como las diferencias de trabajar con Golem-III en Prolog y ASP.

El capítulo 4 describe el escenario que se utilizó para hacer la prueba en Answer Set Programming, donde Golem actúa como empleado de una tienda, así como la base de datos que se utiliza para su implementación y la forma en que se transforma de Prolog a ASP.

El capítulo 5 habla de los resultados obtenidos. Se analiza la complejidad en ambos casos (Prolog y ASP) y se hace una comparativa de tiempo respecto a cuánto le toma a cada sub-módulo generar una respuesta. Por último, se llega a la conclusión que sí es posible replicar el escenario establecido, el cual fue originalmente programado en Prolog y se llegó a una respuesta equivalente. Sin embargo, el costo computacional es demasiado alto para ser usado en una base de datos de dimensión significativa, por lo que no se recomienda usarse en un escenario más complejo.

2 Marco Teórico

2.1 Robot Golem

El grupo Golem comenzó en 1998 bajo la supervisión del Dr. Luis Pineda, tomando como antecedente el proyecto DIME, el cual fue creado con el objetivo de desarrollar tecnología computacional para la construcción de agentes conversacionales multimodales en español hablado.

En 2002 se creó el primer Golem, al integrar el proyecto DIME junto con distintas modalidades de entrada y salida, junto con la conducta motora del robot para la construcción de un agente móvil e inteligente que interactuara en el contexto de una conversación multimodal esquemática.

En 2010 comenzó el proyecto Golem-II, el cual agregó bastantes cambios al antiguo modelo Golem, como el comportamiento reactivo en la navegación y orientación del robot a la fuente de sonido para encarar al interlocutor en una conversación. Además, Golem cambió su estructura para acercarse más a un robot humanoide.

Figura 1. Robot Golem-III

En 2016 estuvo listo el robot Golem-III (Ver Figura 1), el cual compitió ese mismo año en la competencia Robocup, obteniendo el sexto lugar de todos los participantes. Es el modelo que se utiliza en el trabajo presente.

Este robot tiene la capacidad de moverse en su entorno de una manera autónoma por medio de sensores de proximidad para asegurar que no choque con ningún objeto, así como la capacidad de almacenar diferentes ubicaciones por medio de coordenadas.

Además, para poder interactuar con su entorno, cuenta con una cámara que le permite ver y analizar objetos que se encuentren en frente, y

brazos robóticos con los cuales puede tomarlos y soltarlos en diferentes ubicaciones.

2.2 Modelo IOCA

El comportamiento de Golem es regulado por una arquitectura orientada a la interacción objetiva (IOCA, por sus siglas en inglés). Se muestra el diagrama a continuación:

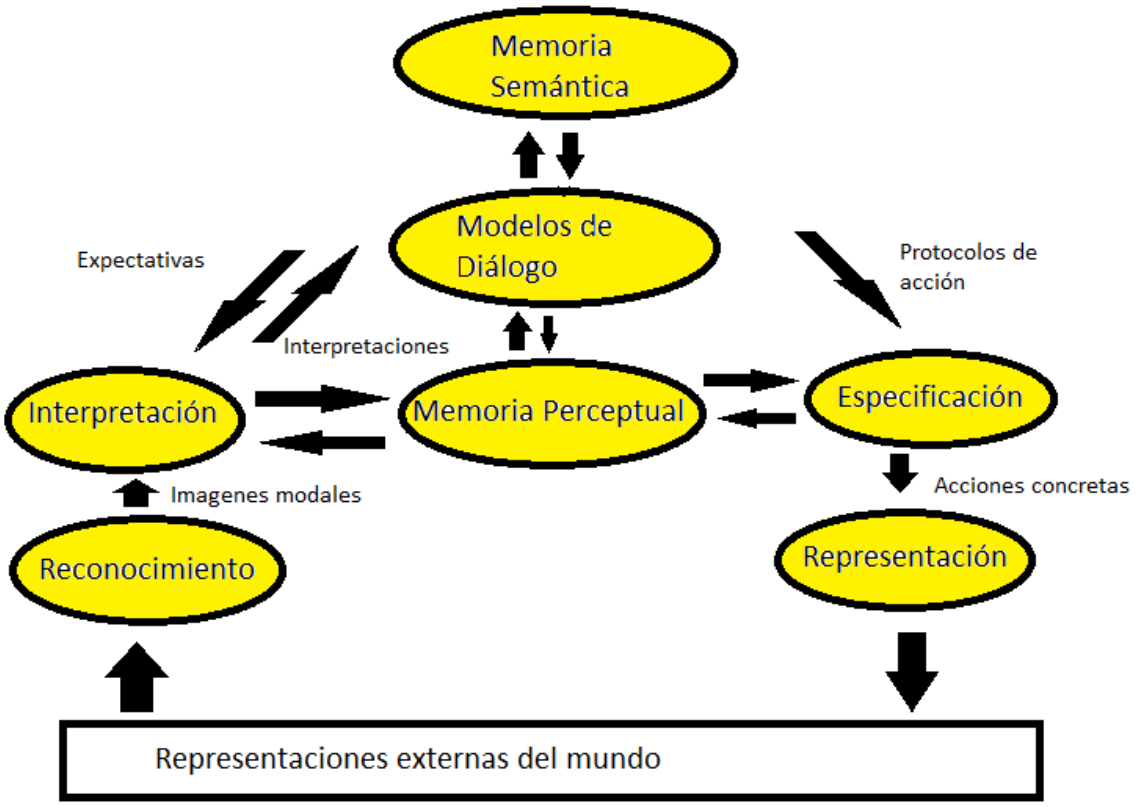


Figura 2. Modelo IOCA traducido de “http://golem.iimas.unam.mx/ioca.php?lang=es”

Los estímulos externos son procesados por los módulos de reconocimiento. Los modelos de diálogo (Capítulo 2.3.1) son el centro de IOCA y describen la tarea a través de un grupo de situaciones.

Después de que un estímulo externo es captado por el módulo de reconocimiento (por ejemplo, una cámara detecta que la persona movió su brazo de cierta manera), el módulo de interpretación le asigna un significado (Por ejemplo, el movimiento del brazo significa que Golem debe seguir a la persona). Esto lo hace según las expectativas del modelo de diálogo y con el módulo de memoria perceptual, pues es el que guarda la información. Una vez que Golem sabe qué es lo que debe hacer, el módulo de especificación se

encarga de generar la información para la acción deseada (en el caso de tener que moverse Golem, necesita las coordenadas a las cuales debe posicionarse) y con esa información el módulo de representación realiza los cambios en el mundo externo (Para poder moverse Golem, los motores que se encargan del movimiento deben girar para llegar a la posición indicada).

Una parte importante de IOCA es que los modelos de diálogo perciben e interactúan con el mundo de una manera abstracta. Se da un significado abstracto al estado del mundo y al conjunto de expectativas posibles que pueden seguir que, en realidad, podría haber sido percibidas por cualquier número de reconocedores. Este paradigma ofrece gran flexibilidad en el desarrollo de software. Así los reconocedores y representadores se vuelven reemplazables y modulares, mientras que la descripción de la tarea permanece intacta. Esto también asegura un marco con las diferentes tareas que se pueden describir y que no requiere de una reescritura completa del software interno. Por otra parte, dado que los reconocedores y representadores son modulares, también se puede reutilizar para diferentes tareas con relativa facilidad.

2.3 SitLog

Programar robots de servicio incluye diferentes tipos de programas que pueden definirse en tres niveles:

1- Los algoritmos para las acciones y percepciones básicas del robot (visión, voz, navegación, etc).
2- Un sistema que se encargue de los procesos y los agentes, comunicación, coordinación y los drivers de los dispositivos de entrada y salida.
3- Representación y programación de la estructura de tareas del robot (ingeniería de comportamiento).

El sistema usado para controlar al robot Golem se llama SitLog y fue creado para la especificación, representación e interpretación de tareas para robots de servicio.

SitLog se encarga de proporcionar la estructura de una tarea en lo que llamaremos “Modelos de diálogo”. SitLog, por medio de los modelos de diálogo, se encarga de decirle a Golem-III los posibles eventos que ocurrirán en el escenario actual y generar las salidas correspondientes (Por medio de IOCA). Esto se puede describir como un gráfico de nodos, donde los nodos representan los eventos dentro del escenario y los vértices las

expectativas y las acciones que realiza Golem-III para llegar a ese nodo. El intérprete de SitLog está escrito en Prolog, por lo que la notación de los modelos de diálogo se parece a este y las funciones internas de las acciones internas también están en Prolog.

2.3.1 Estructura de tareas, situaciones y modelos de diálogo

Las tareas de los robots de servicio se pueden clasificar en estados, en los cuales el robot es capaz de realizar ciertas tareas dependiendo de la información de entrada y posiblemente en los estados visitados previamente. Aquí, los estados se refieren a estados del mundo y a objetos que contienen información en la memoria del robot. Cuando se interpreta el contexto del mundo y el robot "cree" estar en cierto estado, se pueden realizar acciones necesarias para completar cierta tarea. Cuando el robot está fuera de contexto, el robot necesita realizar acciones que lo hagan entrar en contexto antes de realizar la tarea.

¿Cuánta información necesita contener un estado?

- Si los estados tienen poca información y el siguiente estado depende en gran parte de estímulos externos, los cómputos son eficientes, pero tareas complejas son difíciles de modelar.
- Si los estados contienen mucha información y determinar el siguiente estado requiere inferencia, programar la tarea se vuelve sencillo pero a un precio computacional alto.

Definimos una situación como una representación del conjunto de expectativas y acciones potenciales en el contexto de una tarea, así como a la información de control requerida dentro de la estructura de la tarea.

Las expectativas son representaciones de posibles interpretaciones en un contexto dado en la estructura de una tarea, y difieren de la información "cruda" producida por un sensor de bajo nivel que necesita ser controlados por procesos de bajo nivel en un contexto independiente.

De forma más general, las situaciones son objetos que contienen información del contexto y la noción de presuposiciones de que el robot está situado en un contexto en relación con una tarea. Debido a esto, la noción de una situación está restringida a tareas donde está condición se satisface.

2.3.2 Especificaciones de SitLog

Una situación T está representada como un conjunto de modelos de dialogo(DM) **T= [dm1, dm2,...,dmn]**. Así mismo, un DM está constituido por un conjunto de tipos de situaciones dm =[s1, s2,…,sn]. Durante el proceso de interpretación, DMs y situaciones son creadas y ejecutadas dinámicamente, creando un gráfico que corresponde a la estructura de la tarea.

De acuerdo a la estructura de prolog, el nombre de un DM o una situación puede ser una constante o un predicado con uno o más argumentos.

Una situación tiene tres atributos obligatorios:

- Id: un identificador (puede tener una lista de argumentos) para cada situación dentro del DM.
- Tipo: un identificador del tipo de entrada de información (visión, voz) y tiene un algoritmo de interpretación
- Arcs: conjunto de objetos de forma *Expect:action => Next_sit* . El cual representa la expectativa, la acción y la siguiente situación del “arc” del DM.

Además, hay dos tipos genéricos: recursivo y final. Cada DM debe contener una o varias situaciones finales, cada una para una posible conclusión de la tarea y consecuentemente, no tener atributos arcs. El tipo recursivo, es para una situación que contiene un DM.

A continuación se muestra un ejemplo de lo que sería un programa

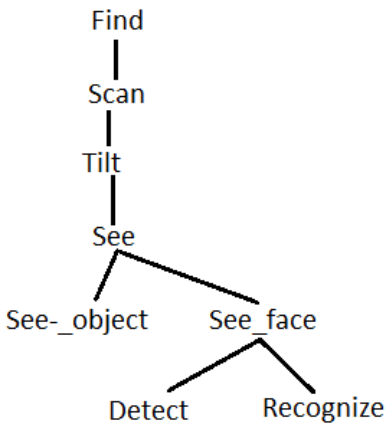
```
diag_mod(ejemplo,
[
        [
                        id ==> is,
                        type ==> neutral,
                        arcs ==> [
                                empty :
                                        [
                                                apply(do_something,[])
                                        ]
                                => fs
                        ]
                ],

                [
      id ==> fs,
      type ==> final
                ]

  ],
).
```

Código 1. Ejemplo de un modelo de diálogo

Y un ejemplo de programa en SitLog:



Los argumentos para la tarea *find* son los siguientes:

1- La entidad a buscar (objeto o persona).

2- El identificador de la persona o lista de objetos. Si la entidad no se encuentra, el MD regresa el primer objeto o lista de objetos encontrado en el proceso de exploración.
3- Una lista de posiciones de búsqueda (el camino de búsqueda).
4- La lista de posiciones horizontales de escaneo que el robot necesita para inspeccionar cada posición de búsqueda.
5- La lista de movimientos verticales que el robot necesita para inspeccionar cada posición de búsqueda.
6- Un modo de observación en caso de que la entidad deseada sea una persona: memorizar o reconocer.
7- La lista de objetos encontrados que fueron especificados como objetos deseados, con su información paramétrica (posición en relación a su posición de búsqueda actual).
8- La lista de posiciones de búsqueda que permanecen sin explorar cuando la tarea *find* haya sido completada, por si el robot está en búsqueda de varios objetos, pueda continuar su búsqueda desde su posición actual después de la primera observación exitosa.
9- El estado de la tarea reportando el éxito de la tarea o el estatus de la última observación hecha en el proceso de búsqueda.

A continuación: se muestra la forma en que quedaría la tarea “find” (De acuerdo a los parámetros mencionados):

```
Find(object, ['lemon tea', gatorade, Pepsi], [pt1,pt2,pt3,pt4], [left, right], [-30, 15], _,
Found_Objects, rest_Positions, Status]
```

Código 2. Ejemplo de una tarea

2.4 Robots de servicio

En la actualidad, existen diferentes robots de servicio que se desenvuelven en diferentes campos de trabajo. Algunos de estos ejemplos son: robots aspiradora, robots quirúrgicos, robots para personas discapacitadas, etc.

Actualmente, la mayoría de estos robots están diseñados para realizar funciones específicas y trabajan en escenarios limitados. Sin embargo, se busca ampliar el campo en el que pueden ser útiles. Por ejemplo, robots contra incendios o robots de escape, los

cuales son dirigidos a zonas de desastre, y por lo tanto cambian de entorno frecuentemente.

Las características principales de estos robots son que no son utilizados en áreas industriales y que funcionan sin un operador humano.

2.5 Planeación

2.5.1 Solución de problemas por búsqueda

En inteligencia artificial, se puede definir planeación como la búsqueda de una secuencia de acciones dirigidas a alcanzar un objetivo. Uno de los métodos que se usan para este objetivo, es el de “Solución de problemas por búsqueda”. Esto significa buscar acciones que lleven al mundo a un estado donde podemos decir que el objetivo se ha cumplido. Existen diferentes algoritmos los cuales se encargan de buscar una solución óptima a diferentes problemas. Algunos de ellos son:

- Depth-First-Search (DFS): En este algoritmo, el problema de representa como un árbol de nodos, en donde cada nodo representa una acción, y una rama del árbol representa el conjunto de acciones a realizar en orden. Para este algoritmo, se expanden cada uno de los nodos de forma recurrente, y cuando se haya alcanzado el final de una rama, el algoritmo hace back-tracking (regresar un nivel atrás en la rama) y analiza el resto de los nodos no analizados. Este proceso se repite hasta que ya no haya más nodos en el árbol.

- Breadth-First-Search (BFS): Similar a DFS, pero este algoritmo revisa todos los nodos de un nivel antes de pasar al siguiente nodo de una rama. Se utiliza para encontrar el camino más corto del nodo raíz a una posible solución. No garantiza encontrar la solución óptima.

- A * : Este algoritmo se utiliza para encontrar la solución óptima de un problema. Cuando los nodos tienen cierto costo, este algoritmo revisa el nodo con el menor costo y genera una rama por cada posible nodo de búsqueda. Se expande el nodo que tenga el menor costo total desde la raíz. El algoritmo se detendrá una vez que llegue a una solución.

2.5.1 Planeación clásica

En inteligencia artificial, los parámetros mínimos que necesita un agente para realizar planeación son: una descripción del estado inicial del mundo, una descripción del objetivo a alcanzar y un conjunto de acciones posibles. Además, de acuerdo a Stuart Russell y Peter Norvig, la planeación clásica cumple con otras características:

"En un problema clásico de planeación, se considera que el mundo es completamente observable, determinístico (No hay aleatoriedad; En cierto estado, ciertas acciones siempre producen el mismo resultado), finito, estático (Los cambios solo se producen cuando un agente realiza una acción) y discreto (Tiene un número contable de estados)." [12]

Aunque planeación en general involucra algún tipo de algoritmo de búsqueda, se necesita también de una buena heurística para evitar la búsqueda de planes que involucren acciones irrelevantes al objetivo, pues la complejidad puede crecer exponencialmente dependiendo del problema, haciendo imposible encontrar una solución en un tiempo viable.

Uno de los algoritmos de planeación basado en estados es el de "forward-reasoning", donde el algoritmo comienza desde un estado inicial, revisa las posibles acciones de acuerdo a sus pre-condiciones y busca diferentes estados hasta llegar al objetivo. Este tipo de búsqueda no toma en cuenta el problema de acciones irrelevantes, por lo que la búsqueda puede llegar a ser innecesariamente larga.

Otro tipo de algoritmo es el "backward-reasoning", donde el algoritmo comienza con el objetivo deseado, busca los estados donde un conjunto de acciones alcancen ese objetivo y así sucesivamente hasta encontrar un estado que coincida con el estado inicial. En este caso, solo se toman en cuenta acciones relevantes. Para este tipo de algoritmo, las acciones deben estar definidas por un conjunto de pre-condiciones, y las acciones deben ser consistentes, es decir, que una acción no deshaga un estado deseado.

2.5.2 Planeación probabilística

Además de la planeación clásica, existen también modelos de planeación probabilística, los cuales generalmente se ocupan cuando no hay observabilidad durante la ejecución del plan. En este tipo de problema, se suele tener una creencia de un estado inicial con cierta

probabilidad entre un conjunto de estados posibles, y una solución posible es el conjunto de estados que alcancen un objetivo deseado con una probabilidad que supere cierto límite dado. [13]

En este tipo de problemas, se pueden usar los algoritmos de búsqueda que se usan para los problemas de planeación clásica, pero se necesita una forma de calcular las probabilidades, por ejemplo, calcular los estados por redes bayesianas, donde realizando un conjunto de acciones, la probabilidad de encontrarme en un segundo estado es la probabilidad de estar en un estado inicial por la probabilidad de que las acciones me lleven a este segundo estado.

2.6 Answer Set Programming

Answer Set Programming (ASP) es un lenguaje declarativo orientado para resolver problemas de búsqueda difíciles (principalmente problemas NP-hard). Es un lenguaje de programación con lógica no-monotónica concebido recientemente por Vladimir Lifschitz [1999,2002] y propuesto por otras personas alrededor del mismo tiempo [Marek y Truszcynski, 1999], [Niemelä, 1999]. Los problemas de búsqueda son reducidos a buscar modelos estables y en principio siempre terminan, a diferencia de Prolog donde puede quedarse ciclado infinitamente.

La forma en que ASP trabaja es generando “modelos estables”, basándose en reglas dado un cierto programa. Las reglas son comandos y condiciones los cuales se cumplen de acuerdo a los parámetros dados. Así mismo, también consiste de “Constraints”, los cuales son condiciones que no deben cumplirse.

2.6.1 Sistema DLV

DLV es un sistema basado en programación lógica disyuntiva creado en conjunto por un equipo italiano-austriaco conformado de investigadores de la Universidad de Calabria y la Universidad Tecnológica de Viena y es el sistema que será usado en este trabajo para correr el código de ASP.

DLV es uno de los sistemas de ASP más usados, y su éxito se debe a varios factores, como por ejemplo:

- Alta expresividad en su lenguaje de representación del conocimiento

- Front-ends para lidiar con aplicaciones específicas

- Técnicas de optimización de búsqueda

- Evaluación paralela

El lenguaje de DLV está compuesto de reglas, las cuales consisten de una cabeza y un cuerpo, donde el cuerpo es una conjunción de funciones y una cabeza es una disyunción de posibles opciones dentro del modelo. Estas opciones se definen como átomos.

Cabeza :- Cuerpo

Una regla dice que si el cuerpo es cierto, entonces la cabeza es cierta. Si una regla no tiene cuerpo, entonces se dice que es un hecho, ya que siempre resulta cierto. Así mismo, si un cuerpo no tiene cabeza, se dice que es una restricción fuerte, y debe ser falsa en cualquier posible solución.

Las soluciones de un problema se llaman “answer sets”. Un programa puede no tener answer sets (No tiene ninguna solución), tener 1 answer set (Tiene una sola solución) o tener varios answer sets (Tiene varias soluciones).

Una característica del sistema DLV es que además de tener restricciones fuertes, las cuales no deben cumplirse bajo ningún motivo, también cuenta con restricciones débiles, las cuales además deben incluir un peso. Al cumplirse una restricción débil, el peso que tenga incluido se añadirá a la solución (El peso de una regla normal se toma como cero) y la solución óptima será el answer set con el menor peso.

2.7 Tareas prácticas y el robot Golem-III

Competencias como el robocup muestran las posibilidades de los robots de servicio. Aunque son pruebas hechas con fines de demostración, muestran simples escenarios involucrando unas cuantas personas, objetos, acciones y eventos designados, limitados en espacio y tiempo, donde el robot necesita completar tareas específicas mientras colabora con otras personas.

Para la construcción de este tipo de aplicaciones, la metodología que se usa aquí es de dos tipos de DMs:

1- Las que modelan la estructura de una tarea como un todo, como servir de mesero en un restaurante
2- Aquellas dirigidas a modelar habilidades generales que pueden ser usadas en diferentes tareas, como aprender la cara de una persona, su nombre, encontrar gente en un cuarto.

3 Metodología

Golem-III está actualmente pensado para desarrollarse como empleado en una tienda o un restaurante, por lo que los programas irán dirigidos a escenarios de este tipo.

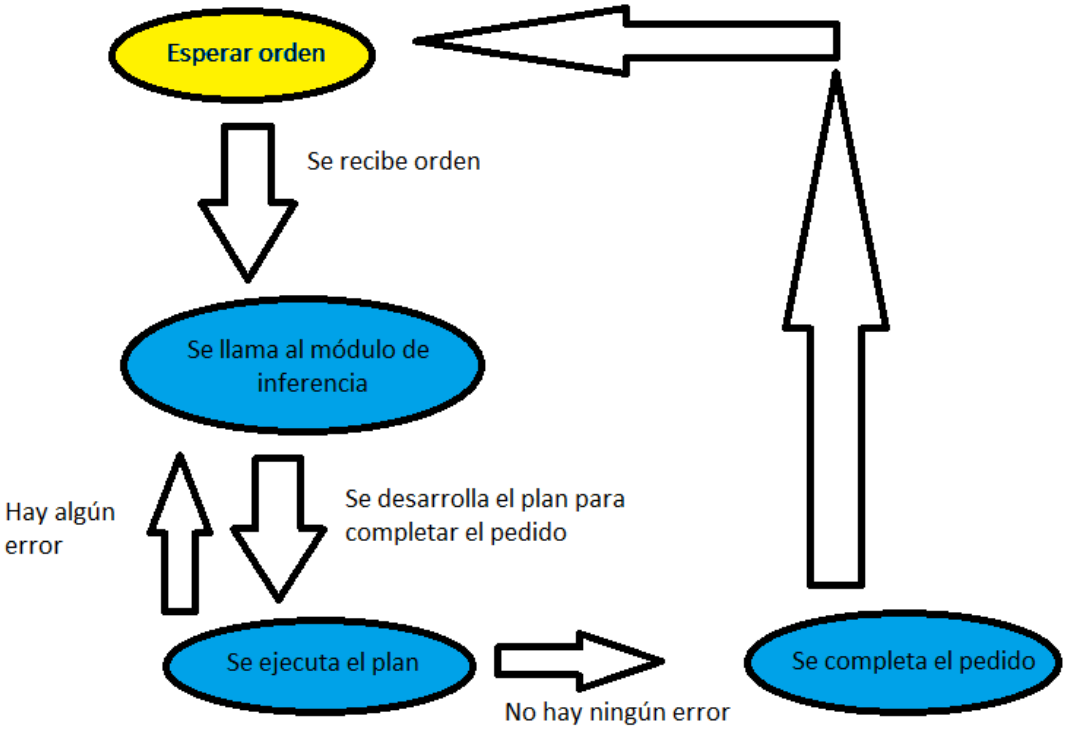


Figura 3. Diagrama del proceso para completar una orden

Para recibir indicaciones de empleados y clientes, Golem-III cuenta con el API de diálogo de Windows (Windows Speech API), el cual le permite recibir estas indicaciones por medio de voz, para después ser procesadas por parte del modelo IOCA del cual se habló en el capítulo 2.

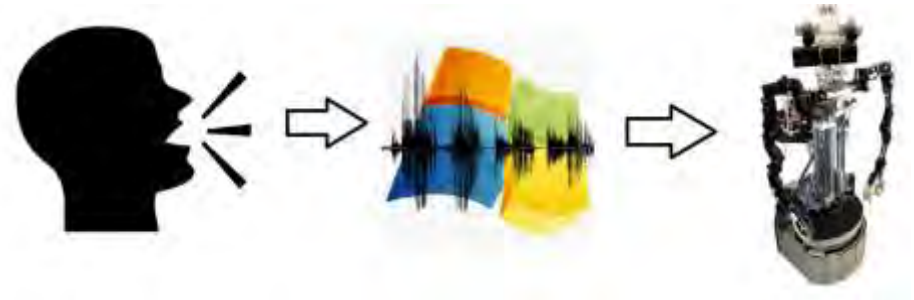


Figura 4. Recibir comandos de usuarios humanos

En el caso de los empleados, los cuales indican los objetos que depositan en las repisas, los comandos que darán serán de tipo: "*I will put…(Objetos)".* Por ejemplo, para indicar que el empleado ha puesto cereal y fideos en las repisas, el comando será: "*I will put cereal and noodles".* No es necesario indicar las repisas, pues Golem-III conoce las repisas correspondientes a los objetos.

En el caso de los clientes, deberán preguntar a Golem-III si el artículo que desean está disponible. Por ejemplo, en caso de querer cereal deberán preguntar a Golem-III: "*Do you have cereal?"*. También es posible hacer preguntas más generales. Por ejemplo, Golem-III tiene cerveza en su base de datos, la cual solo puede ser entregada a clientes mayores de edad. Un cliente podría hacer la pregunta: "*Do you have a drink for all ages?".* En este caso, Golem-III revisaría cuáles artículos registrados tienen las características de ser bebidas y no estar restringidas a una edad específica.

Para resolver los problemas que se presenten, el robot Golem-III tendrá tres módulos que deberán comunicarse entre sí y resolver tareas de distinto nivel. Los tres módulos son:

- Módulo de diagnóstico: En este módulo, el robot hará inferencias sobre el estado del mundo sin que se le diga explícitamente cual es el problema. El mayor uso de este módulo será cuando ocurra un evento inesperado.
- Módulo de toma de decisiones: En este módulo, cuando el robot se encuentre en un estado inesperado y después de haber recibido la causa de esto (enviado desde el módulo de diagnóstico), el robot deberá decidir cuál es la mejor acción a tomar para resolver dicho problema y regresar a un estado deseado en el mundo.
- Módulo de planeación: En este módulo, el robot tiene una lista de acciones a realizar (Ya sea entregada desde el módulo de toma de decisiones o de algún cliente) y deberá decidir de qué forma realizar dichas tareas para obtener la mayor optimización posible.

Por ejemplo: En caso de que Golem-III reciba una orden de un cliente de una soda, se llamarán los tres módulos. El módulo de inferencia no generará ninguna respuesta, puesto que una orden de un cliente no genera ninguna contradicción con ningún dato en la base de datos. El módulo de decisión generará una respuesta de llevar la soda de la repisa donde se encuentre al cliente. El módulo de planeación decidirá que Golem-III debe ir a la

repisa correspondiente, buscar la soda, agarrar la soda, dirigirse al cliente y entregarle la soda.

En este trabajo nos concentraremos en el módulo de planeación y se hablará sobre el módulo de decisión.

Para poder realizar las tareas satisfactoriamente, el robot contará con una base de datos, la cual contendrá la ubicación de todos los lugares a los que podría necesitar dirigirse y una lista de todos los objetos disponibles, así como la ubicación de los mismos. Además, el robot contará con acciones básicas que podrá realizar para poder interactuar con el mundo. Estas acciones son:

- Moverse: Este comando se utiliza para que el robot se desplace de un lugar a otro recibiendo una coordenada [X,Y,Z], donde "X" es la posición horizontal, "Y" es la posición vertical y "Z" es el ángulo de rotación del robot. Se toma como la posición inicial [0,0,0] la ubicación y ángulo de rotación que tenía Golem al ser encendido.
- Escanear: Este comando permite al robot usar la cámara para buscar objetos que se encuentran enfrente de él. Este comando se usará para buscar los productos que ha solicitado el cliente en las repisas correspondientes.
- Agarrar: Este comando se utiliza para agarrar un objeto una vez que Golem lo haya captado con su cámara y haya deducido su ubicación con relación a él mismo.
- Entregar: Este comando se usa para que el robot suelte algún objeto que tiene en alguna de sus manos. En este caso se usa para poder entregar las órdenes del cliente y reacomodar objetos que estén fuera de lugar.

3.1 Módulo de decisión

El módulo de decisión (Anexos II) revisa los objetos que se encuentran actualmente en el mundo (según la base de datos) y las órdenes que tiene registradas. Golem-III tiene en su base de datos la ubicación en la que deberían encontrarse los objetos (refrescos en la repisa de bebidas, fideos en la de comida, etc.), y si encuentra alguna discrepancia, decide que el objeto debe cambiarse de lugar, siempre y cuando el objeto no esté registrado como una orden del cliente.

Por ejemplo: si Golem descubriera que el refresco está en la repisa equivocada, el razonador le diría que debe reacomodarlo en la repisa de refrescos. Sin embargo, en caso de tener registrado que el cliente ordenó el refresco, deberá llevarlo con el cliente, sin importar la repisa en la que se encuentre.

La respuesta que entrega el razonador es lo que recibirá el módulo de planeación para llegar a la solución óptima. Debido a que el sistema DLV utiliza archivos para extraer información y el sistema ya implementado en Golem-III necesita la respuesta dentro del código en forma de variable, el razonador entregará la solución en forma de lista dentro de una variable, mientras que al mismo tiempo genera un archivo para ser llamado por el módulo de planeación.

Una vez que se llama el módulo de decisión, el programa se corre dos veces: la primera entrega la respuesta con las variables “bring(Object, Location)” y “realign(Object, Location)”. Esta salida se guarda para la función de Golem que se encarga de decir en voz alta la respuesta generada. La segunda vez que se llama la función, la respuesta generada es en variables “moveFromTo(Object, CurrentLocation, DesiredLocation, N)”. Esta segunda respuesta se guarda en un archivo y es la que será llamada por el módulo de planeación para generar el plan correspondiente. La razón por la que se usan dos respuestas diferentes es porque Golem ya tiene un formato exacto con el cual espera sus respuestas, mientras que ASP necesita tener una variable “N” la cual indica el paso en la cual se genera cierta acción.

3.1.1 Detalles técnicos del módulo de decisión

El módulo de decisión (Anexos II) creará los átomos “location(Objeto, Lugar, ‘observed’)” y “location_helper(bring(Objeto, Lugar)”, pues solo dependen de la base de datos. Location_helper dependerá de los objetos que algún empleado anteriormente haya indicado fueron puestos en las repisas correspondientes, o de las órdenes que algún cliente haya realizado.

A partir del átomo “location_helper”, obtenemos el átomo “location(Objeto, Lugar, ‘target’)”. Esto servirá para comparar entre el mundo observado y el mundo deseado.

El estado inicial de este módulo será el mundo observado (la creencia de en donde se encuentran los objetos) y el estado final será el mundo deseado (llevar los objetos con el cliente, o acomodar un objeto en su debida repisa, en caso de no encontrarse ahí).

La transición del estado inicial al estado final es directa, ya que no depende de ninguna acción ni de ninguna otra variable. Es decir, el módulo de decisión toma cualquier discrepancia entre un estado y otro y los toma como objetivos para el módulo de planeación.

Location (Objeto, Lugar_1, "observed") ---------- Location (Objeto, Lugar_2, "target")

Para cada objeto, checa si su lugar observado y su lugar deseado son iguales (siempre y cuando estén disponibles en la base de datos). En caso de no serlo, la respuesta de este módulo será que debe mandar el objeto del "Lugar_1" al "Lugar_2".

3.2 Respuesta para Golem-III

Como se mencionó anteriormente, Golem espera una respuesta con las variables "bring(Object, Location)" y "realign(Object,Location)". El módulo de decisión generará una respuesta de tipo "b_bring(Object,Location)" y "b_realign(Object,Location)". Esto es debido a que la función en Prolog que utiliza Golem necesita que las respuestas sean de tipo compound, mientras que al generarlas en ASP se generan como String, por lo que es necesario modificarlas un poco con las funciones auxiliares (Anexos V) antes de entregarlas.

Golem generará la respuesta con el formato:

*[Best Model:{**b_bring(X1,Y1), b_realign(X2,Y2), …**} ,Cost ([Weight:Level]): <0>]*

Por medio de varias funciones en Prolog (Anexos V), se tomará únicamente la parte deseada (señalada en negritas) para terminar en una lista de la siguiente manera:

[bring(X1,Y1), realign(X2,Y2), …]

Las variables dentro de esta lista están como compounds, por lo que Golem podrá utilizarla en futuras funciones.

3.2.1 Respuesta para el módulo de planeación

La segunda vez que el programa corre, ASP genera una respuesta de tipo:

[Best Model:{moveFromTo(X1,Y1,Z1,N), moveFromTo(X2,Y2,Z2,N), …} ,Cost ([Weight:Level]): <[N:1]>]

Para quedarnos únicamente con la respuesta, hacemos algo similar a lo que se hizo para la respuesta anterior. Lo que nos queda es:

moveFromTo(X1,Y1,Z1,N), moveFromTo(X2,Y2,Z2,N), …

Esta respuesta no será llamada directamente por ninguna función, sino que será escrita en un archivo para ser llamada por el módulo de planeación, por lo que no se necesita hacer modificaciones a las variables. Lo único que hace falta es darle una cabeza para que ASP lo pueda leer como un átomo. La respuesta que se escribirá en el archivo será:

objective(N):- moveFromTo(X1,Y1,Z1,N), moveFromTo(X2,Y2,Z2,N), #int(N).

3.3 Módulo de planeación

El módulo de planeación realiza los diferentes posibles movimientos de Golem en cierto momento determinado, ejecutando una sola acción a la vez. Estas acciones son moverse, agarrar objeto, soltar objeto, buscar objeto y no hacer nada. Dentro del programa se tienen diversas reglas a seguir, y restricciones a respetar, las cuales son:

- Golem debe haber buscado un objeto para poder agarrarlo
- Golem debe tener agarrado un objeto para poder soltarlo
- Golem debe tener una ubicación en todo momento
- En caso de haber encontrado un objeto en cierto momento, Golem debe intentar agarrarlo en el momento siguiente
- Golem puede tener, a lo mucho, dos objetos agarrados en cualquier momento
- Una vez que se hayan cumplido todos los objetivos, Golem deberá no hacer nada por el tiempo restante
- Golem no puede encontrarse en dos lugares a la vez
- Golem no puede estar en un lugar diferente al que estaba anteriormente sin haber realizado una acción de moverse
- Golem no puede encontrarse en la misma posición en la que se encontraba anteriormente después de haber realizado la acción de moverse
- Golem no puede soltar dos objetos a la vez
- Golem no puede no hacer nada en cierto momento y después realizar una acción

Así mismo, se tienen ciertas restricciones a seguir para evitar que el programa realice sets que no pueden contener ninguna solución. Estas restricciones son:

- Golem debe encontrarse en la misma posición que un objeto para intentar buscarlo
- Golem no puede realizar dos acciones de moverse inmediatamente una después de otra
- Golem no puede intentar agarrar un objeto sin haberlo buscado antes
- Golem no puede realizar la acción de soltar un objeto si no tiene ningún objeto agarrado

Aunque es posible realizar los objetivos únicamente con el primer conjunto de reglas y restricciones, el segundo conjunto ayuda a minimizar el número de sets generados en el momento de buscar la solución óptima. Esto es importante, ya que con cada posible set, la cantidad de memoria usada incrementa exponencialmente, y sin hacer reducción de sets de respuestas "inútiles", el programa podría quedarse sin memoria antes de encontrar cualquier solución, dependiendo del número de pasos que se necesiten.

Una vez que la respuesta sea generada y procesada para tener el formato que Golem espera, quedará de la siguiente forma:

[move(position, Destination), search(Object), grasp(Object), move…]

Esta lista se mandará como una variable al modelo de diálogo ya implementado en Golem el cual se encarga de que Golem realice las acciones sobre el mundo.

3.3.1 Detalles técnicos del módulo de planeación

El módulo de planeación (Anexos III) recibe la base de datos transformada para ASP y el query generado por el módulo de decisión, para saber cuáles son los objetivos que debe cumplir.

Al comenzar el módulo, el mundo se encuentra en un estado inicial, el cual depende completamente de la base de datos, ya que tanto Golem como los objetos se encuentran donde la base lo indica.

Para cada momento "N", se cumplirá un átomo diferente de entre "move", "search", "grab" y "drop". Estos átomos representarán las acciones que se realizan en cada momento.

El estado final que se busca es aquel donde los objetos pasan del lugar donde se encuentran en su estado inicial a donde indica que deben estar el query generado por el módulo de decisión.

La forma en que los objetos pasarán de un estado a otro es por medio de los predicados “seen”, “grabbed” y “dropped”.

Un objeto se sabe que fue agarrado por medio del átomo “grabbed”, el cual se cumple si Golem realiza la acción “grab” en cierto tiempo, que en el tiempo anterior se cumpla el átomo “seen”, y que tenga menos de dos objetos agarrados.

El átomo “seen” se cumple si Golem realiza la acción “search” en cierto tiempo, y Golem se encuentra en el mismo lugar que un objeto.

El átomo “dropped” se cumple si Golem tiene un objeto agarrado y realiza la acción “drop” en cierto tiempo.

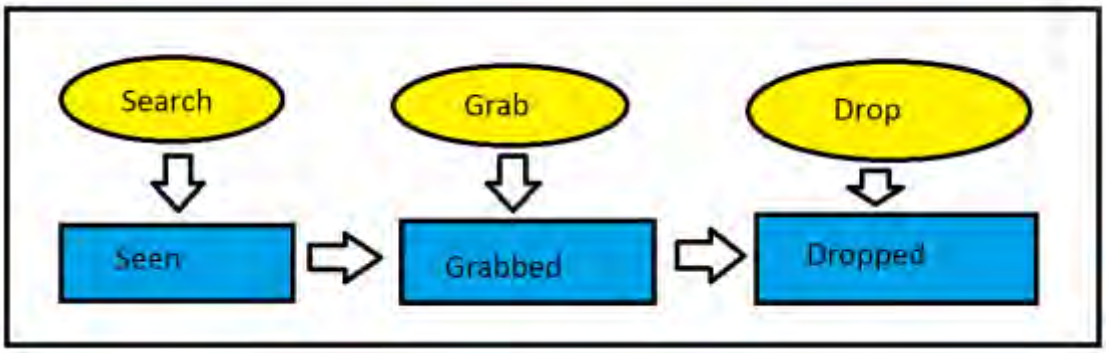


Figura 5. Representación de las acciones y los estados de los objetos

Es decir, para que un objeto pase de una ubicación a otra, se deben cumplir los siguientes predicados:

golem_place (ubicacion_objeto), seen (Objeto), grabbed (Objeto), golem_place (Ubicación_deseada), dropped(Objeto)

Esto se hace por medio de las siguientes acciones:

move(ubicacion_objeto), search(objeto), grab (objeto), move(ubicacion_deseada), drop(objeto)

Estas acciones no necesariamente deben realizarse una directamente tras otra, pero sí deben llevarse en el orden correcto.

Para conocer las ubicaciones de los objetos, se contarán con los átomos “product_place”, los cuales indican que los objetos se encuentran donde dice la base de datos, y cambiarán únicamente si se cumplen los átomos “grabbed” y “dropped”.

3.4 Trabajar con Prolog vs trabajar con ASP en Golem-III

Golem trabaja originalmente con Prolog, por lo que el formato de la base de datos está pensado para este mismo. Las variables en la base de datos están ordenadas en listas, lo que lo hace difícil para trabajar directamente en ASP, ya que este trabaja con “Nesting”, y una lista puede ser indefinidamente larga, con lo que el programa no puede estar seguro de tener suficiente memoria para conseguir una respuesta.

Prolog trabaja con “backward-reasoning”, lo que significa que primero tiene la meta y de ahí checa si es posible de acuerdo a los átomos que se tienen definidos, y ASP trabaja con “forward-reasoning”, lo que significa que primero toma la información conocida, y de ahí infiere información nueva de acuerdo a las reglas dadas por el programa hasta que se alcance el objetivo deseado.

Una ventaja de ASP sobre Prolog es que no importa el orden en que se escriban las reglas o los objetivos en el programa, mientras que en Prolog sí. Además, ASP tiene la opción de dar negación fuerte (Se dice explícitamente que hay una característica negada) y negación débil (No puedo inferir con la información que tengo que una característica sea cierta), mientras que Prolog únicamente tiene negación débil.

En cuanto a la implementación del código, y como se mencionó brevemente en el módulo de decisión, el sistema DLV necesita que la información sea proporcionada por medio de archivos, mientras que el programa en Prolog utiliza variables. Primero, el diagnosticador lee la base de datos, genera una solución sobre qué es lo que pasó y crea un archivo convirtiendo la base de datos de listas a átomos para que lo pueda leer DLV en el módulo de decisión (Todo esto lo hace sin destruir la base de datos en Prolog). El módulo de planeación no necesita generar un archivo para depositar la respuesta, ya que es el último módulo en ser llamado, así que se recibe en forma de variable para que el resto del programa pueda leerlo. Además, Debido a las diferencias entre el formato de Prolog y

ASP, es necesario pasar las diferentes listas de la base de datos en Prolog a forma de átomos para que puedan ser leídos por el solver de ASP.

Una de las mayores diferencias entre trabajar en ASP y Prolog es la forma en que la información está organizada en la base de datos. En Prolog la información está contenida en forma de listas de la siguiente manera:

[class(nombre, Padre,[Propiedades], [Relaciones],[Instancias])]

Estas listas deben pasar a átomos para ser leídas por ASP.

Por ejemplo: En Prolog tendríamos la lista para cereal de la siguiente manera

```
class(cereal,food,[inv=>0],[],[[id=>c1,[brand=>kellogs,original_id=>c1,in=>storage],[]]])
```

Aquí se indica que la clase es cereal, su padre es comida, tiene la propiedad “inventario” con un valor de 0, no tiene relaciones y tiene una instancia con un identificador “id” igual a c1, cuyas propiedades son que la marca es kellogs, su “id_original” es c1 y se encuentra en el almacén.

La equivalencia de esta clase en el código de ASP sería de la siguiente manera:

```
class("cereal").
parent("cereal",food).
property("cereal",inv,0).
instance("cereal",id,c1).
property(c1,brand,kellogs).
property(c1,original_id,c1).
property(c1,in,storage).
```

4 Experimento

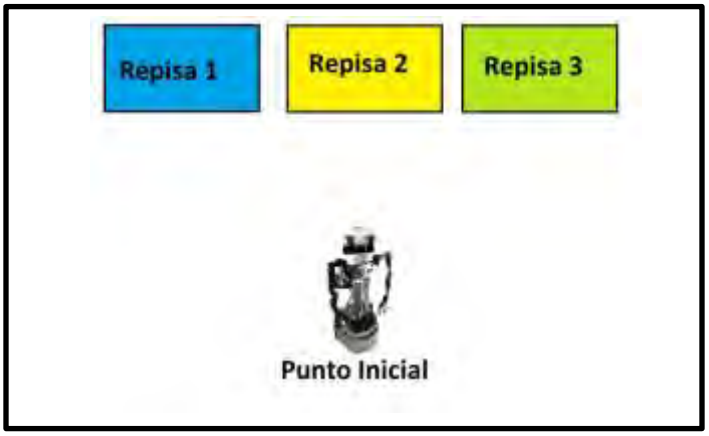


Figura 6. Diagrama físico del escenario

4.1 Escenario

El escenario usado para esta prueba está pensado para poder demostrar las capacidades técnicas de Golem-III, donde el robot toma el papel de empleado de una tienda encargado de recibir órdenes de los clientes.

Golem-III debe ser capaz de recibir indicaciones de los empleados sobre los objetos depositados en las repisas, así como órdenes de los clientes. Para poder registrar los comandos de los usuarios humanos, tanto empleados como clientes, los objetos a los que se hacen referencia deben estar en la base de datos de Golem-III.

En este escenario, un empleado le informa a Golem que ha puesto en los estantes una coca, una cerveza, unos fideos y un bollo. Golem lo registra en su base de datos y sabe que la coca y la cerveza deben ir en la repisa de las bebidas (repisa 1), los fideos deben ir en la repisa de la comida (repisa 2) y el bollo en la repisa de los panes (repisa 3).

Para indicar esto, el empleado da la indicación verbal: "*I will put a coke, a Heineken, noodles and biscuits".*

El primer llamado el módulo de inferencia ocurre cuando el cliente le pide a Golem-III un refresco. Primero Golem hace su diagnóstico asumiendo que el empleado depositó los objetos en las repisas correctas. El módulo de decisión decide que debe llevarle la soda al cliente, y el módulo de planeación decide que debe ir a la repisa 1, buscar el refresco, agarrar el refresco, regresar a la posición de inicio y entregarle el refresco al cliente.

En este caso, el cliente le hace la pregunta a Golem-III: "*Do you have a drink for all ages?" ("*¿Tienes alguna bebida para todas las edades?"). Golem-III responde: "*Yes, coke. I will bring it to you".* ("Sí, coca. Te la traeré).

Figura 7. Cliente pidiendo un refresco a Golem-III

En el momento de ir a la repisa 1, se da cuenta que el mundo no está como él pensaba, pues el refresco no se encuentra ahí, además de que los fideos se encuentran en esa repisa cuando deberían estar en la repisa 2. En ese momento vuelve a llamar al módulo de inferencia. El módulo de diagnóstico llega a la conclusión de que puso los fideos en la repisa 1 y el refresco en la repisa 2. El módulo de decisión decide que debe acomodar los fideos y llevar la orden al cliente. El módulo de planeación decide que el camino más óptimo para hacer esto es buscar los fideos, agarrar los fideos, ir a la repisa 2, dejar los fideos, buscar el refresco, agarrar el refresco, ir a la posición de inicio y entregar el refresco al cliente.

Figura 8. Golem-III va a la repisa 1 y se da cuenta que no está el refresco

Una vez que llega a la repisa 2, se da cuenta que el cereal se encuentra ahí (cuando eso no lo tenía en su base de datos). Golem actualiza la base de datos y continúa con su plan. Deja los fideos en la repisa pero se da cuenta que no está el refresco. En ese momento vuelve a llamar al módulo de inferencia. El módulo de diagnóstico llega a la conclusión de que el empleado puso el cereal en la repisa 2 y el refresco en la repisa 3. Dado que los fideos ya fueron acomodados en su lugar, el módulo de decisión únicamente decide que debe llevar el refresco al cliente. El módulo de planeación decide que debe ir a la repisa 3, buscar el refresco, agarrar el refresco, ir a la posición de inicio y entregar el refresco al cliente.

El refresco se encuentra en la repisa 3 y el objetivo se ha cumplido exitosamente.

Figura 9. Golem-III encuentra el refresco en la repisa 3

4.2 Implementación del escenario

Como el objetivo de este trabajo es hacer un análisis sobre la posibilidad del uso de ASP en un escenario que ya se encuentra implementado en Prolog, se usarán los mismos modelos de diálogo y funciones ya implementadas. Lo único que cambiará será el módulo

de inferencia: más específicamente, los módulos ya mencionados anteriormente (Diagnóstico, decisión y planeación).

Para poder decir que es posible realizar el escenario, los módulos deben ser capaces de entregar las mismas salidas que en Prolog bajo las mismas circunstancias en las que se probaron originalmente. Además de esto, debido a que hay otras funciones que dependen de estas salidas (Por ejemplo, al llamar al módulo de inferencia, este dice en voz alta lo que pasó, lo que piensa y hacer y cómo lo piensa hacer), el resultado de ASP debe no solamente ser una equivalencia al resultado del programa en Prolog, sino que además debe tener el mismo formato.

4.3 Módulo de planeación en Prolog

El módulo de planeación en Prolog originalmente implementado en Golem trabaja con un algoritmo DFS (Depth First Search), el cual crea un árbol de acciones (cada nodo del árbol representa una acción), recorre cada posible nodo del árbol y se queda con la mejor opción disponible.

El algoritmo crea un estado inicial, donde se indica que Golem se encuentra en la posición inicial y tiene ambas manos libres. Después toma los comandos dados por el razonador y los descompone en sub-objetivos, los cuales consisten en listas de acciones que se deben completar para haber realizado la acción. Por ejemplo, para llevar un objeto de un lugar a otro, las acciones que se deben realizar son ir al lugar donde se encuentra el objeto, buscar el objeto, agarrar el objeto, ir a la posición deseada y soltar el objeto. Además de esto, se toma un tiempo máximo para completar el plan deseado. Este tiempo es entregado por el usuario humano al llamar la función en el módulo de inferencia.

Después, el algoritmo crea una lista de posibles acciones. Cada acción tiene condiciones que se deben cumplir para poder entrar a esta lista:

- Para entregar un objeto, Golem debe encontrarse en la posición donde se desea dejar el objeto (de acuerdo a los sub-objetivos generados) y tener el objeto en la mano

- Para buscar un objeto, Golem debe estar en la misma posición que un objeto (de acuerdo a la base de datos)

- Para agarrar un objeto, Golem debe estar en la misma posición que un objeto, haberlo encontrado antes y tener una mano libre

- Para moverse a un lugar donde debe dejar un objeto, debe haber un sub-objetivo donde se desee dejar un objeto en tal lugar, y Golem debe encontrarse en una posición diferente

- Para dejar un objeto, Golem debe tenerlo en la mano

Una vez que se tienen las posibles acciones, Golem escoge la primera y continúa expandiendo los nodos (las posibles acciones). En cada nodo, se guarda la probabilidad de haber completado todas las acciones hasta el momento, la recompensa que se ha generado, el costo por haber llegado hasta ese nodo (el cual será representado por el tiempo) y un valor "G" igual a:

G = probabilidad * recompensa * Tiempo_de_bonus.

Este tiempo de bonus por:

$Tiempo\_bonus = 1 + (T_{max} - Costo) / (2 * T_{max})$

Donde "Costo" es el tiempo que toma realizar todas las acciones generadas hasta el momento y "T_{max}" es igual al tiempo máximo permitido para completar los objetivos.

El algoritmo seguirá expandiendo nodos, elegirá el primero de estos nodos y repetirá el proceso hasta que se complete una de las siguientes condiciones:

1- Todos los sub-objetivos hayan sido completados
2- El costo de las acciones haya superado el Tiempo máximo permitido

Una vez que se cumpla una de estas condiciones, el algoritmo regresará un nodo atrás y en caso de haber nodos sin explorar, los revisará uno por uno repitiendo el proceso. Una vez que haya regresado y comience a revisar nodos alternos, seguirá expandiendo nodos hasta completar alguna de las dos condiciones previamente mencionadas, o bien, si el valor "G" en cierto nodo sea mayor al conseguido previamente. Esto significaría que el nodo encontrado no puede ser una solución mejor a la ya encontrada previamente y no tiene caso seguirlo recorriendo.

Cada vez que el algoritmo encuentre un nodo con una solución, lo comparará (de acuerdo a su valor “G”) con la solución previamente encontrada y conservará únicamente el que tenga un mayor valor.

4.4 Base de datos

Como se mencionó en el capítulo anterior, la base de datos debe ser transformada de listas a átomos para que pueda ser leída por el programa en ASP. Esto se hace con una función en Prolog (Anexos IV) y funciona recorriendo cada clase de la lista y generando los átomos uno por uno. Sabemos que las clases en la base de datos están en el siguiente orden:

Class(nombre, padre, [Propiedades], [Relaciones], [Instancias, [Propiedades], [Relaciones]])

Como sabemos la posición exacta dentro de cada clase, podemos crear el átomo “class” para la variable de la primera posición, el átomo “parent” para la variable de la segunda posición, y así sucesivamente.

Los átomos que se generan son los siguientes:

Class(nombre).

Parent(nombre, padre).

Property(nombre, propiedad, valor).

Relation(relación, clase_1, clase_2).

Instance(nombre_clase, “id”, id_instancia).

Al inicio del escenario, la base de datos se encuentra de la siguiente forma:

```
[class(top,none,[],[],[]),

class(object,top,[],[],[]),

class(comestible,object,[graspable,not(on_discount)],[],[]),

class(food,comestible,[shelf=>shelf2],[],[]),

class(cereal,food,[inv=>0],[],[[id=>c1,[brand=>kellogs,in=>storage,original_id=>c1],[]]]),

class(instant food,food,[inv=>0],[],[[id=>i1,[brand=>noodles,in=>storage,
original_id=>i1],[]]]),
```

Código 3. Base de datos en Prolog

```
(Aquí se muestra solo una parte de la base de datos. Para ver el resto,
ir al anexo VI).
```

Esto se puede ver representado en el siguiente diagrama, donde los círculos son las clases y los cuadrados las instancias:

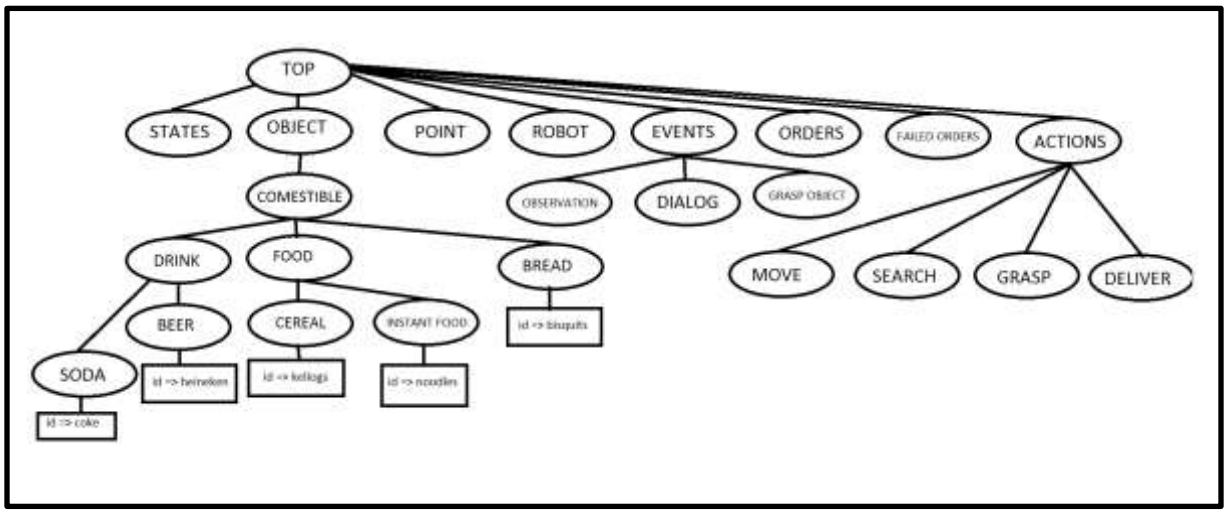


Figura 10. Diagrama de la base de datos

Y al pasarla a átomos para ASP, se transforma de la siguiente manera:

```
class("comestible").
parent("comestible",object).
property("comestible",graspable,null).
-property("comestible",on_discount,null).

class("food").
parent("food",comestible).
property("food",shelf,shelf2).

class("cereal").
parent("cereal",food).
property("cereal",inv,0).
instance("cereal",id,c1).
property(c1,brand,kellogs).
property(c1,original_id,c1).
property(c1,in,storage).

class("instant food").
parent("instant food",food).
property("instant food",inv,1).
instance("instant food",id,i1).
property(i1,brand,noodles).
property(i1,original_id,i1).
property(i1,in,storage).
```

Código 4. Base de datos en ASP

```
(Aquí se muestra solo una parte de la base de datos. Para ver el resto,
ir al anexo VII).
```

Al separar las diferentes características de cada clase, puede ser mucho más fácil para un usuario humano interpretar las propiedades de los diferentes objetos en ASP que en una lista gigante como originalmente se encuentra en Prolog. A lo largo del escenario, lo único que puede cambiar en la base de datos es la posición de los objetos, por lo que solo cambiaría el átomo “property(nombre, “in”, lugar)”.

5 Resultados

5.1 Complejidad del módulo de planeación en ASP

Para poder resolver el problema, el módulo de planeación de ASP utiliza cabezas disyuntivas para la ubicación del golem, el estado de los objetos (agarrados y vistos) y la acción que se debe realizar en cada tiempo:

grab(Time) v drop(Time) v search(Time) v move(Time) v idle(Time) :- #int(Time).

golem_place(X,T) v notGolemPlace(X,T):- possible_locations(X), #int(T).

dropped(X,Y,N) v notDropped(X,Y,N):- golem_place(Y,N), grabbed(X,N1), drop(N1), N=N1+1.

seen(X,N) v notSeen(X,N):- golem_place(Y,N), product_place(X,Y,N), search(N1), N=N1+1.

El primer conjunto de reglas disyuntivas consiste en las acciones que puede realizar Golem en cada momento. En cada momento, el programa decidirá realizar una acción, y de ahí revisará las consecuencias que esto tiene en el mundo. Como en cada tiempo puede realizar cualquier acción (Con excepción de “idle”, debido a las restricciones implementadas), el número máximo de sets generados es igual a 4^N, donde “N” es el número de pasos realizados antes de haber encontrado una solución. Sin embargo, estos sets no se guardan todos en memoria al mismo tiempo, sino que al generar el primer set, se guarda junto con el peso total producido. Después se genera otro set y se compara el peso total de este con el previamente guardado y de ahí se guarda el de menor peso. Este proceso se repite hasta haber inspeccionado todos los posibles sets, por lo que en cada momento se guardan 2 sets en la memoria.

En cada set generado, Golem necesita indicar, para cada ubicación, si se encuentra o no se encuentra ahí. Para los objetos solo se necesita indicar dónde se encuentran.

El estado de los objetos (vistos y agarrados) solo se cumple cuando el Golem se encuentra en la ubicación de los objetos y realiza la acción de ver o soltar, por lo que no se generan en cada paso.

Esto significa que para cada set generado, la memoria que se utiliza es igual a:

$$O(N * Obj + Lug * N) = O(N * (Obj + Lug))$$

N: Número de pasos

Obj: Cantidad de objetos en la base de datos

Lug: Ubicaciones posibles registradas en la base de datos

Y la memoria utilizada para todo el programa es igual a:

$$O(2 * N * (Obj + Lug))$$

5.2 Complejidad del módulo de planeación en Prolog

Debido a que el algoritmo que se utiliza es DFS, la complejidad de espacio es igual a :

$$O(V)$$; donde V es el número de nodos generados

Debido a que existen cuatro acciones diferentes, los nodos máximos generados por cada nivel en el árbol son igual a 4. Como en cada nivel del árbol no se expanden todos los nodos al mismo tiempo, sino que se revisan uno por uno, el número de nodos máximos que tendremos en cada momento es igual a 4 * N, donde “N” es igual al nivel del árbol. El nivel del árbol será igual al número de pasos realizados hasta el momento.

El número de pasos depende de la cantidad de acciones que debe realizar Golem para completar el objetivo generado por el módulo de decisión, y del tiempo máximo permitido.

Con esto vemos que este algoritmo incrementa la memoria usada de forma lineal, en vez de exponencialmente como lo hace ASP, por lo que es más eficiente en términos de memoria.

5.2 Resultados de tiempo para Prolog y ASP

Podemos observar en la siguiente tabla los tiempos que se necesitan para cada módulo en cada llamada al módulo de inferencia a lo largo de todo el escenario:

	Diagnóstico	**decisión**	**Planeación**
Primer llamado: Principio del escenario. No existe ningún error ni orden que completar	Prolog: 233 ms	Prolog: 659 ms	Prolog: 23 ms
	ASP: 66 ms	ASP: 44 ms	ASP: 121 ms
Segundo llamado: Golem recibe una orden del cliente por una coca	Prolog: 2012 ms	Prolog: 928 ms	Prolog: 151 ms
	ASP: 70 ms	ASP: 51 ms	ASP: 96 ms
Tercer llamado: La coca no es encuentra en la primera repisa	Prolog: 1287 ms	Prolog: 1332 ms	Prolog: 249 ms
	ASP: 76 ms	ASP: 63 ms	ASP: 618 ms
Cuarto llamado: La coca no es encuentra en la segunda repisa	Prolog: 1553 ms	Prolog: 841 ms	Prolog: 131 ms
	ASP: 181 ms	ASP: 56 ms	ASP: 352 ms

Como se ha visto, el escenario planteado se ha podido replicar en ASP logrando conseguir el mismo resultado que en Prolog sin diferencias significativas de tiempo para planeación (aunque sí hay mejoras para diagnóstico y planeación), por lo que podemos decir que ASP tiene el potencial para trabajar con robots de servicio.

6 Conclusiones

Después de haber desarrollado este trabajo, las conclusiones a las que se llegaron son las siguientes:

La hipótesis de que ASP es capaz de modelar planeación para un robot de servicio es correcta, ya que el escenario se ha podido replicar de una manera similar tanto en Prolog como en ASP. Las respuestas que se consiguieron en ambas implementaciones fueron idénticas, así que Golem-III no pierde eficiencia en sus módulos implementados en ASP. Asimismo, gracias a la alta expresividad del sistema DLV, los códigos generados son no solamente más fáciles de entender para un usuario humano, sino que también son más cortos que sus contrapartes en Prolog.

Sin embargo, debido a que ASP tiene un incremento en el uso de memoria -el cual depende de los objetos, lugares y el número de pasos necesarios- no es viable usarse para bases de datos de dimensión significativa donde se usa un número elevado de objetos y/o de ubicaciones. Además, considerando que el módulo de inferencia en ASP sólo es una implementación diferente a Prolog, no agrega ninguna acción extra a Golem-III, por lo que la ventaja que podría ganar es el tiempo y el riesgo sería el no tener suficiente memoria para crear un plan. Debido a que no hay forma de reducir la memoria utilizada -pues por la forma en que trabaja ASP Golem-III debe tener registrado dónde se encuentra y dónde no se encuentra en cada instante en el tiempo- esta implementación sólo puede usarse en escenarios pequeños. Por ejemplo, pruebas donde la intención sea demostrar las capacidades de Golem y no dónde el objetivo sea colocarlo en una situación real como sería un minisúper.

A pesar de sus limitaciones, ASP ha sido implementado en varios proyectos, como soporte de decisión en el transbordador STS, por lo que vemos que ASP puede resultar útil en proyectos donde no se necesite multiplicar la información varias veces.

Con el desarrollo del módulo de inferencia, así como la comparación entre dos implementaciones diferentes, nos damos cuenta de la complejidad del pensamiento humano, pues todos estos pasos una persona los hace de forma inconsciente en todo tipo de situaciones. Podemos ver entonces, que muchos procesos humanos, los cuales parecen sencillos, requieren una gran cantidad de información y un pensamiento complejo.

Bibliografía

[1] The Golem Group. (2011). Arquitectura orientada a la interacción cognitiva. 11 diciembre 2016, de Grupo golem Sitio web: http://golem.iimas.unam.mx/ioca.php

[2] Vladimir Lifschitz. (July 13 - 17, 2008). What is answer set programming?. AAAI'08 Proceedings of the 23rd national conference on Artificial intelligence , Volumen 3, Paginas 1594-1597.

[3] Brewka, Gerhard, Thomas Eiter, and Mirosław Truszczyński. "Answer Set Programming At A Glance". Communications of the ACM 54.12 (2011). 11 Dec. 2016.

[4] Luis A. Pineda, Lisset Salinas, Ivan V. Meza, Caleb Rascon and Gibran Fuentes. SitLog: A Programming Language for Service Robot Tasks. *Int J Adv Robot Syst*, 2013, 10:358. doi: 10.5772/56906

[5] Luis A. Pineda, Arturo Rodríguez, Gibran Fuentes, Caleb Rascon and Ivan V. Meza. Concept and Functional Structure of a Service Robot. *Int J Adv Robot Syst*, 2015, 12:6. doi: 10.5772/6002

[6] Alvano Mario, Faber Wolfgang, Leone Nicola, Perri Simona, Pfeifer Gerald, Terracina Giorgio. (March 16 - 19, 2010). The disjunctive datalog system DLV. Datalog'10 Proceedings of the First international conference on Datalog Reloaded, Paginas 282-301.

[7] Eiter Thomas, Giovambattista Ianni, Thomas Krennwallner. (2009). Answer Set Programming: A Primer. En Reasoning Web. Semantic Technologies for Information Systems (Páginas 40-110). Berlin: Springer Berlin Heidelberg.

[8] Magdalena Ortiz, Mantas Šimkus. (2012). Reasoning and Query Answering in Description Logics. En Reasoning Web. Semantic Technologies for Advanced Query Answering (Paginas 1-53). Berlin: Springer Berlin Heidelberg.

[9] Eiter Thomas, Faber Wolfgang, Leone Nicola, Gerald Pfeifer, Axel Polleres. (March 2003). A logic programming approach to knowledge-state planning, II: the DLV system. *Artificial Intelligence*, Volumen 144, Paginas 157 - 211 .

[10] Eiter Thomas, Faber Wolfgang, Nicola Leone, Gerald Pfeifer. (enero 1999). The Diagnosis Frontend of the dlv system. *AI Communications*, Volumen 12, Paginas 99 - 111

[11] Eiter T., Faber W., Leone N., Pfeifer G., Polleres A. (2002) Answer Set Planning under Action Costs. In: Flesca S., Greco S., Ianni G., Leone N. (eds) Logics in Artificial Intelligence. JELIA 2002. Lecture Notes in Computer Science, vol 2424. Springer, Berlin, Heidelberg

[12] Russell, S. & Norvig, P. (2003). Artificial intelligence a modern approach (2nd ed.). New Jersey: Prentice Hall.

[13] Carmel Domshlak, Jorg Hoffmann. (2006). Fast Probabilistic Planning Through Weighted Model Counting. ICAPS'06 Proceedings of the Sixteenth International Conference on International Conference on Automated Planning and Scheduling, Paginas 243-252 .

Anexos

I- Módulo de inferencia

Este módulo se manda a llamar cuando Golem se encuentra con algún error.

```
inference_module:-
    open_kb(KB),
    etr_dlv_link(KB,Diagnostic,NewKB),
    etr_save_kb(NewKB),
    decision_maker_ASP(Decision),
    dfs_planner_ASP(Plan),
    ddp_generate_explication_dialog(Diagnostic,Decision,Plan,Dialog),
    assign_func_value(solve_plan(Plan,Dialog)).
```

II- Módulo de decisión

Este módulo se encarga de generar la salida que le dirá al módulo de planeación qué es lo que debe hacer. Las salidas que crea son de tipo bring(X,Y) para usarse en el módulo de inferencia o de tipo moveFromTo(X, Lugar1, Lugar 2, N) para que el módulo de planeación lo pueda leer.

```
location(X,Y,observed):- property(Z,in,Y), property(Z,brand,X), instance(_,id,Z).

location_helper(X):- property(dialog1,content,Z), #length(Z,Len), #int(Ext), Ext<=Len, #getnth(Z,Ext,X).

location(X,Y,target):- location_helper(bring(X,Y)).


loc(X,Y,W,0):- location(X,Y,W).


change_place(N) v idle(N):- #int(N).

bringHelper(X,Y,start,N):- instance("orders",id,bring(X)), loc(X,Y,observed,N), change_place(N), X!=nothing.

bringHelper(X,Y,Z,N):- loc(X,Y,observed,N), loc(X,Z,target,N), change_place(N), Y!=Z, X!=nothing, not bringHelper(X,Y,start,N).


moveFromTo(W,Y,Z,"N"):- bringHelper(X,Y,Z,0), property(W,brand,X).

b_bring(X,client):- bringHelper(X,_,start,N).
```

```
b_realign(X,Z):- bringHelper(X,_,Z,N), Z!=start.


loc(X,Y,target,N2):- loc(X,Y,target,N), N2=N+1.

loc(nothing,Y,observed,N2):- bringHelper(X,Y,_,N), N2=N+1, not loc(Q,Y,observed,N),
Q!=X, loc(Q,_,_,_).

loc(X,Y,observed,N2):- loc(X,Y,observed,N), N2=N+1, not bringHelper(X,Y,Z,N),
loc(_,Z,_,_).

loc(X,Z,observed,N2):- bringHelper(X,Y,Z,N), N2=N+1.


:- change_place(N2), idle(N1), N2=N1+1.

:~ not loc(X,Y,observed,1), loc(X,Y,target,1).[1:1]
```

III- Módulo de planeación

Este módulo se encarga de generar la salida de acuerdo a la información en la base de datos y al objetivo que haya generado el módulo de decisión. Las salidas que genera son de tipo action(N, acción, Complemento).

```
grab(Time) v drop(Time) v search(Time) v move(Time) v idle(Time) :- #int(Time).


motion(Time):- grab(Time).

motion(Time):- drop(Time).

motion(Time):- search(Time).

motion(Time):- move(Time).


grabbed_objects(0,0).

grabbed_objects(C,N2):- grabbed_objects(C,N), not grab(N), not drop(N), N2=N+1.


grabbed_objects(C,N2):- grabbed_objects(C1,N), grabbed(_,N2), grab(N), N2=N+1,
C=C1+1.

grabbed_objects(C,N2):- grabbed_objects(C1,N), dropped(_,_,N2), N2=N+1, C=C1-1.


golem_place(X,0):- property(golem,position,X).
```

```
golem_place(X,T) v notGolemPlace(X,T):- possible_locations(X), #int(T).

has_location(T):- #int(T), golem_place(X,T).


product_place(X,Y,0):- location_(X,Y), X!=golem, X!=nothing.

product_place(X,Y,N2):- product_place(X,Y,N), N2=N+1, not grabbed(X,N).

product_place(X,Y,N):- dropped(X,Y,N).

product_place(X,hand,N):- grabbed(X,N), not dropped(X,Y,N), golem_place(Y,N).

location(Z,Y,observed):- property(Z,in,Y), property(Z,brand,X), instance(_,id,Z).

location_(X,Y):- location(X,Y,observed).

possible_locations(Y):- location_(_,Y).

possible_locations(start).


grabbed(X,T2):- grabbed(X,T), T2=T+1, not dropped(X,Y,T2), golem_place(Y,T2).

grabbed(X,T2):- grab(T1), seen(X,T1), T2=T1+1, grabbed_objects(C,T1), C<2.


dropped(X,Y,N) v notDropped(X,Y,N):- golem_place(Y,N), grabbed(X,N1), drop(N1),
N=N1+1.


seen(X,N) v notSeen(X,N):- golem_place(Y,N), product_place(X,Y,N), search(N1),
N=N1+1.


:- golem_place(Pos1,T1), golem_place(Pos2,T2), T2 = T1+1, Pos1!=Pos2, not move(T1).

:- golem_place(Pos1,T), golem_place(Pos2,T), Pos1!=Pos2.

:- golem_place(Pos,T), golem_place(Pos,T2), move(T), T2=T+1.

:- #int(T), not has_location(T).

:- seen(X,T), seen(Y,T), X!=Y, #int(T).

:- dropped(X,Z,T), dropped(Y,Z,T), X!=Y, #int(T).

:- move(T1), move(T2), T2=T1+1.

:- grab(T), not search(T1), T=T1+1, #int(T1).

:- search(T), not grab(T1), T1=T+1.
```

```
:- drop(T), grabbed_objects(0,T).

:- idle(T), motion(T2), T2=T+1.

:~ not idle(T), #int(T).[1:1]

:- idle(N), not objective(N).

:- objective(N), not idle(N).


action(N, drop, Object):- drop(N), dropped(Object,_,N2), N2=N+1.

action(N, grab, Object):- grab(N), grabbed(Object,N2), not grabbed(Object,N), N2=N+1.

action(N, search, Object):- search(N), seen(Object,N2), N2=N+1.

action(N, move, Destination):- move(N), golem_place(Destination,N2), N2=N+1.


moveFromTo(X,Y,Z,N):- grabbed(X,N1), dropped(X,Z,N), N=N1+1, location_(_,Y).

moveFromTo(X,Y,Z,N2):- moveFromTo(X,Y,Z,N), N2=N+1.


:- not objective(#maxint).


:- deliver(N2), grasp(N1), N2=N1+1.

:~ grab(N), #int(N).[410:1]

:~ drop(N), #int(N).[5:1]

:~ search(N), #int(N).[500:1]

:~ move(N), #int(N).[500:1]
```

IV- Transformar base de datos de Prolog a ASP (Código desarrollado por Ricardo Adolfo Fierro Villaneda y adaptado por Eduardo Tello Ramos)

```
:- op(800,xfx,'=>').

% Transforms Golems database into a format that can be used by DLV.
etr_create_file_for_aspdlv([],_).
etr_create_file_for_aspdlv([class(X,actions,P,R,I)|Z],Stream):-
    write(Stream,'parent("'), write(Stream,X), write(Stream,'",'),
write(Stream,actions), write(Stream,').'), nl(Stream),

write(Stream,'special_class('),write(Stream,X),write(Stream,',actions,"'),write
```

```
(Stream,P),write(Stream,'",'"'),write(Stream,R),write(Stream,'",'"'),write(Strea
m,I),write(Stream,'").'), nl(Stream),
    etr_create_file_for_aspdlv(Z,Stream),
    !.
etr_create_file_for_aspdlv([class(X,Y,P,R,I)|Z],Stream):-
    write(Stream,'class("'), write(Stream,X), write(Stream,'").'),  nl(Stream),
    write(Stream,'parent("'), write(Stream,X), write(Stream,'",'),
write(Stream,Y), write(Stream,').'), nl(Stream),
    etr_write_properties(X,P,Stream),
    etr_write_relationships(X,R,Stream),
    etr_write_instances(X,I,Stream),
    nl(Stream),
    etr_create_file_for_aspdlv(Z,Stream),
    !.

% Writes a classes properties into a file.
etr_write_properties(_,[],_).
% Negated properties
etr_write_properties(X,[not(H)|T],Stream):-
    etr_transform_to_string(H,Z),
    etr_name_value(Z, H1,H2),
    write(Stream,'-property("'), write(Stream,X), write(Stream,'",'),
write(Stream,H1), write(Stream,','), write(Stream,H2), write(Stream,').'),
nl(Stream),
    etr_write_properties(X,T,Stream),
    !.
etr_write_properties(X,[not(H)|T],Stream):-
    write(Stream,'-property("'), write(Stream,X), write(Stream,'",'),
write(Stream,H), write(Stream,',null).'),  nl(Stream),
    etr_write_properties(X,T,Stream),
    !.
% Non negated properties
etr_write_properties(X,[(H)|T],Stream):-
    etr_transform_to_string(H,Z),
    etr_name_value(Z, H1,H2),
    write(Stream,'property("'), write(Stream,X), write(Stream,'",'),
write(Stream,H1), write(Stream,','), write(Stream,H2), write(Stream,').'),
nl(Stream),
    etr_write_properties(X,T,Stream),
    !.
etr_write_properties(X,[H|T],Stream):-
    write(Stream,'property("'), write(Stream,X), write(Stream,'",'),
write(Stream,H), write(Stream,',null).'),  nl(Stream),
    etr_write_properties(X,T,Stream),
    !.

% Writes a classes relationships into a file
etr_write_relationships(_,[],_).
% Negated Relationships
etr_write_relationships(X,[not(H)|T],Stream):-
    etr_transform_to_string(H,Z),
    etr_name_value(Z, H1,H2),
```

```
    write(Stream,'-relation("'), write(Stream,X), write(Stream,'",'),
write(Stream,H1), write(Stream,','), write(Stream,H2), write(Stream,').'),
nl(Stream),
    etr_write_relationships(X,T,Stream),
    !.
etr_write_relationships(X,[not(H)|T],Stream):-
    write(Stream,'-relation("'), write(Stream,X), write(Stream,'",'),
write(Stream,H), write(Stream,',null).'),  nl(Stream),
    etr_write_relationships(X,T,Stream),
    !.
% Non negated relationships
etr_write_relationships(X,[(H)|T],Stream):-
    etr_transform_to_string(H,Z),
    etr_name_value(Z, H1,H2),
    write(Stream,'relation("'), write(Stream,X), write(Stream,'",'),
write(Stream,H1), write(Stream,','), write(Stream,H2), write(Stream,').'),
nl(Stream),
    etr_write_relationships(X,T,Stream),
    !.
etr_write_relationships(X,[H|T],Stream):-
    write(Stream,'relation("'), write(Stream,X), write(Stream,'",'),
write(Stream,H), write(Stream,',null).'),  nl(Stream),
    etr_write_relationships(X,T,Stream),
    !.

% Writes a classes instances into a file
etr_write_instances(_,[],_).
% Sole instance
etr_write_instances(X,[[H|[P,R]]],Stream):-
    etr_transform_to_string(H,Z),
    etr_name_value(Z, H1,H2),
    write(Stream,'instance("'), write(Stream,X), write(Stream,'",'),
write(Stream,H1), write(Stream,','), write(Stream,H2), write(Stream,').'),
nl(Stream),
    etr_write_instance_properties(H2,P,Stream),
    etr_write_instance_relationships(H2,R,Stream),
    !.
% Multiple instances
etr_write_instances(X,[[H|[P,R]]|T2],Stream):-
    etr_transform_to_string(H,Z),
    etr_name_value(Z, H1,H2),
    write(Stream,'instance("'), write(Stream,X), write(Stream,'",'),
write(Stream,H1), write(Stream,','), write(Stream,H2), write(Stream,').'),
nl(Stream),
    etr_write_instance_properties(H2,P,Stream),
    etr_write_instance_relationships(H2,R,Stream),
    etr_write_instances(X,T2,Stream),
    !.

% Writes an instances properties into a file
etr_write_instance_properties(_,[],_).
% Negated instance properties
etr_write_instance_properties(X,[not(H)|T],Stream):-
    etr_transform_to_string(H,Z),
```

```
    etr_name_value(Z, H1,H2),
    write(Stream,-property(X,H1,H2)), write(Stream,'.'),  nl(Stream),
    etr_write_instance_properties(X,T,Stream),
    !.
etr_write_instance_properties(X,[not(H)|T],Stream):-
    write(Stream,-property(X,H,null)), write(Stream,'.'),  nl(Stream),
    etr_write_instance_properties(X,T,Stream),
    !.
% Non negated instance properties
etr_write_instance_properties(X,[H|T],Stream):-
    etr_transform_to_string(H,Z),
    etr_name_value(Z, H1,H2),
    H1 = name,
    write(Stream,'property('),  write(Stream,X),  write(Stream,','),
write(Stream,H1), write(Stream,',"'), write(Stream,H2), write(Stream,'").'),
nl(Stream),
    etr_write_instance_properties(X,T,Stream),
    !.
etr_write_instance_properties(X,[H|T],Stream):-
    etr_transform_to_string(H,Z),
    etr_name_value(Z, H1,H2),
    not(H1 = name),
    write(Stream,property(X,H1,H2)), write(Stream,'.'),  nl(Stream),
    etr_write_instance_properties(X,T,Stream),
    !.
etr_write_instance_properties(X,[H|T],Stream):-
    write(Stream,property(X,H,null)), write(Stream,'.'),  nl(Stream),
    etr_write_instance_properties(X,T,Stream),
    !.

% Writes an instances properties into a file
etr_write_instance_relationships(_,[],_).
% Negated Instance Relationships
etr_write_instance_relationships(X,[not(H)|T],Stream):-
    etr_transform_to_string(H,Z),
    etr_name_value(Z, H1,H2),
    write(Stream,-relation(X,H1,H2)), write(Stream,'.'),  nl(Stream),
    etr_write_instance_relationships(X,T,Stream),
    !.
% Non Negated Instance Relationships
etr_write_instance_relationships(X,[H|T],Stream):-
    etr_transform_to_string(H,Z),
    etr_name_value(Z, H1,H2),
    write(Stream,relation(X,H1,H2)), write(Stream,'.'),  nl(Stream),
    etr_write_instance_relationships(X,T,Stream),
    !.

% Transforms an atom into a string
etr_transform_to_string(Z,A):- with_output_to(atom(A), write(Z)),
    !.

etr_name_value(String, Name, Value) :-
    sub_string(String, Before, _, After, '=>'),
    sub_string(String, 0, Before, _, NameString),
```

```
    atom_string(Name, NameString),
    sub_string(String, _, After, 0, Value),
    !.
```

V- Funciones auxiliares

Esta función es llamada cuando se necesita hacer decisión

```
decision_maker_ASP(Decision):-
    dlvLink_reasoning_asp(Decision).
```

Esta función es llamada cuando se necesita hacer planeación

```
dfs_planner_ASP(Plan):-
    dlvLink_asp(Plan).
```

Esta función es la que manda llamar al sistema DLV junto con los archivos correspondientes. Después manda llamar a la función "create plan asp" para que la acomode en el formato correcto y esa es la salida que se entrega.

```
dlvLink_asp(Result):-
        process_create('apps/robocup_2015/ricardo/dlv.i386-linux-elf-
static.bin', ['-silent', '-N=20', '-n=1', '-filter=action',
'apps/robocup_2015/ricardo/planning_module.txt',
'apps/robocup_2015/ricardo/ETR_A_Prolog_To_ASP.dl',
'apps/robocup_2015/ricardo/query.txt'], [stdout(pipe(B))]),
        read_lines(B, Salida),
        create_plan_asp(Salida, Result).
```

Esta parte se corre en caso de tener un plan vacio. Recibe la salida que genera el sistema dlv con el módulo de planeación y entrega una lista vacía

```
create_plan_asp(Plan, []):-
open('apps/robocup_2015/ricardo/planRealizado.txt',write,Stream),
    write(Stream,Plan),
    close(Stream),
    open('apps/robocup_2015/ricardo/planRealizado.txt',read,Stream2),
    read_stream_to_codes(Stream2, Querycodes),
    close(Stream2),
```

```
    atom_codes(Planbetter,Querycodes),
    string_concat('[Best model: {' , Result, Planbetter),
    string_concat(EndResult, ',Cost ([Weight:Level]): <[0:1]>]', Result),
    open('apps/robocup_2015/ricardo/planRealizado.txt',write,Stream5),
    write(Stream5,'[]'),
    close(Stream5).
```

Esta parte se corre en caso de no tener un plan vacio. Recibe la salida que genera el sistema dlv con el módulo de planeación y entrega una lista ordenada del plan generado.

```
create_plan_asp(Plan, Final_res):-
open('apps/robocup_2015/ricardo/planRealizado.txt',write,Stream),
    write(Stream,Plan),
    close(Stream),
 open('apps/robocup_2015/ricardo/planRealizado.txt',read,Stream2),
    read_stream_to_codes(Stream2, Querycodes),
    close(Stream2),
    remove_lastN(Querycodes,10,Planbetter),
    string_concat('[Best model: {' , Result, Planbetter),
    check_string_concat(Result,Endresult),
    atomic_list_concat(X,', ',Endresult),
    action_in_order(X,0,Fin),
    open('apps/robocup_2015/ricardo/planRealizado.txt',write,Stream3),
    write(Stream3,Fin),
    close(Stream3),
    open('apps/robocup_2015/ricardo/planRealizado.txt',read,Stream4),
    read_stream_to_codes(Stream4, Querycodes2),
    atom_codes(Ordered, Querycodes2),
    close(Stream4),
    string_concat('[',L,Ordered),
    string_concat(L2,']',L),
    atomic_list_concat(X2, '),', L2),
    actions_last_fix(X2,0,Final_res),
    open('apps/robocup_2015/ricardo/planRealizado.txt',write,Stream5),
    write(Stream5,Final_res),
    close(Stream5).
```

Esta función es la que manda llamar al sistema DLV junto con los archivos correspondientes para el módulo de decisión. Después manda llamar a la función "create query for inference" para que la acomode en el formato correcto y esa es la salida que se entrega. Además genera un archivo para el módulo de planeación, pues es más fácil para dlv leer un archivo que una variable. Esto último lo hace con la función "create query asp".

```
dlvLink_reasoning_asp(Decision2):-
                process_create('apps/robocup_2015/ricardo/dlv.i386-linux-elf-
static.bin', ['-silent', '-N=20', '-n=1', '-filter=moveFromTo',
'apps/robocup_2015/ricardo/reasoning_module.txt',
'apps/robocup_2015/ricardo/ETR_A_Prolog_To_ASP.dl',
'apps/robocup_2015/ricardo/prolog_to_asp_target.txt'], [stdout(pipe(B))]),
        read_lines(B, Salida),
        create_query_asp(Salida, Decision),
            process_create('apps/robocup_2015/ricardo/dlv.i386-linux-elf-
static.bin', ['-silent', '-N=20', '-n=1', '-filter=b_bring,b_realign',
'apps/robocup_2015/ricardo/reasoning_module.txt',
'apps/robocup_2015/ricardo/prolog_to_asp.txt',
'apps/robocup_2015/ricardo/prolog_to_asp_target.txt'], [stdout(pipe(C))]),
        read_lines(C, Salida2),
        create_query_for_inference(Salida2, Decision2).
```

Esta función recibe la salida del sistema dlv en el módulo de decisión y la ajusta de tal forma que pueda ser leída por el módulo de planeación. Esta salida es mandada a un archivo

```
create_query_asp(Query,Z):-
    open('apps/robocup_2015/ricardo/query.txt',write,Stream),
    write(Stream,Query),
    close(Stream),
    open('apps/robocup_2015/ricardo/query.txt',read,Stream2),
    read_stream_to_codes(Stream2, Querycodes),
    close(Stream2),
    remove_lastN(Querycodes,10,Querybetter),
```

```
    string_concat('[Best model: {' , Result, Querybetter),
    check_string_concat(Result,Endresult),
    remove_chars(Endresult, '"', Z, 1, N),
    open('apps/robocup_2015/ricardo/query.txt',write,Stream3),
    write(Stream3,'objective(N):-'),
    write(Stream3,Z),
    write(Stream3,',#int(N).'),
    close(Stream3).
```

Esto se llama en caso de que no haya nada en el decisión. La salida es una lista vacía

```
create_query_asp(Query,[]):-
    open('apps/robocup_2015/ricardo/query.txt',write,Stream),
    write(Stream,Query),
    close(Stream),
    open('apps/robocup_2015/ricardo/query.txt',read,Stream2),
    read_stream_to_codes(Stream2, Querycodes),
    close(Stream2),
    atom_codes(Querybetter,Querycodes),
    string_concat('[Best model: {' , Result, Querybetter),
    string_concat(EndResult, '},Cost ([Weight:Level]): <0>]', Result),
    open('apps/robocup_2015/ricardo/query.txt',write,Stream3),
    write(Stream3,'objective(N):- #int(N).'),
    close(Stream3).
```

Esta función recibe la salida del Sistema dlv en el módulo de decisión y la acomoda de tal forma que pueda ser registrada por la función ya implementada en el Golem para la salida original en Prolog.

```
create_query_for_inference(Query,ResultFinal):-
    open('apps/robocup_2015/ricardo/query_for_inference.txt',write,Stream),
    write(Stream,Query),
    close(Stream),

open('apps/robocup_2015/ricardo/query_for_inference.txt',read,Stream2),
    read_stream_to_codes(Stream2, Querycodes),
```

```
    close(Stream2),
    remove_lastN(Querycodes,10,Querybetter),
    string_concat('[Best model: {' , Result, Querybetter),
    check_string_concat(Result,Endresult),
    remove_chars(Endresult, '"', Z, 1, N),
    string_concat('b_' , Z2, Z),
    atomic_list_concat(Z3,', b_',Z2),
    rafv_make_list_of_compounds(Z3,ResultFinal),

open('apps/robocup_2015/ricardo/query_for_inference.txt',write,Stream3),
    write(Stream3,'['),
    write(Stream3,Z2),
    write(Stream3,']'),
    close(Stream3).
```

Esta función recibe la salida del Sistema dlv en el módulo de decisión y se manda a llamar si no hay ninguna decisión. La salida es una lista vacía

```
create_query_for_inference(Query,[]):-
    open('apps/robocup_2015/ricardo/query_for_inference.txt',write,Stream),
    write(Stream,Query),
    close(Stream),

open('apps/robocup_2015/ricardo/query_for_inference.txt',read,Stream2),
    read_stream_to_codes(Stream2, Querycodes),
    close(Stream2),
    atom_codes(Querybetter,Querycodes),
    string_concat('[Best model: {' , Result, Querybetter),
    string_concat(EndResult, ',Cost ([Weight:Level]): <0>]', Result),

open('apps/robocup_2015/ricardo/query_for_inference.txt',write,Stream3),
    write(Stream3,'[]'),
    close(Stream3).
```

Las siguientes funciones se usan para convertir la salida de la decisión de un String a un compound. Esto se hace para que Golem lo pueda registrar.

```
rafv_make_list_of_compounds(Z3,ResultFinal):-
    [H|T] = Z3,
    rafv_make_list_of_compounds_helper(H,T,ResultFinal).

rafv_make_list_of_compounds_helper(H,[],[bring(X1,X2)]):-
    string_concat('bring', Comp_var, H),
    string_concat('(', Comp_var2, Comp_var),
    string_concat(Comp_var3, ')', Comp_var2),
    atomic_list_concat(Final_pair,',',Comp_var3),
    Final_pair = [X1,X2].

rafv_make_list_of_compounds_helper(H,[],[realign(X1,X2)]):-
    string_concat('realign', Comp_var, H),
    string_concat('(', Comp_var2, Comp_var),
    string_concat(Comp_var3, ')', Comp_var2),
    atomic_list_concat(Final_pair,',',Comp_var3),
    Final_pair = [X1,X2].

rafv_make_list_of_compounds_helper(H,T,[bring(X1,X2)|ResultFinal]):-
    string_concat('bring', Comp_var, H),
    string_concat('(', Comp_var2, Comp_var),
    string_concat(Comp_var3, ')', Comp_var2),
    atomic_list_concat(Final_pair,',',Comp_var3),
    Final_pair = [X1,X2],
    [H2|T2] = T,
    rafv_make_list_of_compounds_helper(H2,T2,ResultFinal).

rafv_make_list_of_compounds_helper(H,T,[realign(X1,X2)|ResultFinal]):-
    string_concat('realign', Comp_var, H),
    string_concat('(', Comp_var2, Comp_var),
    string_concat(Comp_var3, ')', Comp_var2),
    atomic_list_concat(Final_pair,',',Comp_var3),
    Final_pair = [X1,X2],
    [H2|T2] = T,
    rafv_make_list_of_compounds_helper(H2,T2,ResultFinal).
```

Las siguientes funciones reciben la salida generada por el Sistema dlv para el módulo de planeación. Debido a que en la salida incluye el costo, queremos eliminar esa parte y quedarnos únicamente con las acciones

```
check_string_concat(X,Y):-
        string_concat(Y, '},Cost ([Weight:Level]):', X).

check_string_concat(X,Y):-
        string_concat(Y, '},Cost ([Weight:Level]): ', X).

check_string_concat(X,Y):-
        string_concat(Y, '},Cost ([Weight:Level])', X).
```

Las siguientes funciones toman la salida generada por el Sistema dlv para el módulo de planeación y le quitan los últimos caracteres. Esto es debido a que no se sabe de antemano cuál será el peso del plan óptimo, y por lo tanto no sabemos qué números estarán incluidos

```
remove_lastN(X,N,Y):-
    remove_last_helper(X,N,Q),
    atom_codes(Y,Q).

remove_last_helper(X,0,X).
remove_last_helper(X,N,Y):-
    list_butlast(X,X2),
    N>0,
    N2 is N-1,
    remove_last_helper(X2,N2,Y).
```

```
get_n_chars([H|T], N, [H|Y]):-
    N2 is N-1,
    get_n_chars(T,N2,Y).

get_n_chars(X,0,[]).
```

Las siguientes funciones reciben el plan en desorden, lo acomodan y lo entregan en un formato que Golem puede entender

```
action_in_order(String_to_read, N, [H|Rest_result]):-
    String_to_read=[H|T],
    atom_codes(H,Q),
    get_n_chars(Q,9,W),
    atom_codes(Head_action,W),
    atom_concat(N,',', Head_helper),
    atom_concat('action(', Head_helper, Head_action),
    N2 is N+1,
    action_in_order(T,N2,Rest_result).
action_in_order(String_to_read, N, [H|Rest_result]):-
    String_to_read=[H|T],
    atom_codes(H,Q),
    get_n_chars(Q,10,W),
    atom_codes(Head_action,W),
    atom_concat(N,',', Head_helper),
    atom_concat('action(', Head_helper, Head_action),
    N2 is N+1,
    action_in_order(T,N2,Rest_result).
action_in_order(String_to_read, N, Result):-
```

```
    String_to_read=[H|T],
    append(T,[H],Z),
    action_in_order(Z,N,Result).
action_in_order([],N,[]).



actions_last_fix(String_to_read, N, [grasp(Helper3)|Rest]):-
    String_to_read =[H|T],
    T = [],
    atom_concat('action(', N, Helper),
    atom_concat(Helper,Helper2,H),
    atom_concat(',grab,',Helper4,Helper2),
    atom_concat(Helper3,')',Helper4),
    N2 is N+1,
    print(H),
    actions_last_fix(T,N2,Rest).
actions_last_fix(String_to_read, N, [move(position,Helper3)|Rest]):-
    String_to_read =[H|T],
    T = [],
    atom_concat('action(', N, Helper),
    atom_concat(Helper,Helper2,H),
    atom_concat(',move,',Helper4,Helper2),
    atom_concat(Helper3,')',Helper4),
    N2 is N+1,
    actions_last_fix(T,N2,Rest).
actions_last_fix(String_to_read, N, [search(Helper3)|Rest]):-
    String_to_read =[H|T],
```

```
    T = [],
    atom_concat('action(', N, Helper),
    atom_concat(Helper,Helper2,H),
    atom_concat(',search,',Helper4,Helper2),
    atom_concat(Helper3,')',Helper4),
    N2 is N+1,
    actions_last_fix(T,N2,Rest).
actions_last_fix(String_to_read, N, [deliver(Helper3)|Rest]):-
    String_to_read =[H|T],
    T = [],
    atom_concat('action(', N, Helper),
    atom_concat(Helper,Helper2,H),
    atom_concat(',drop,',Helper4,Helper2),
    atom_concat(Helper3,')',Helper4),
    N2 is N+1,
    actions_last_fix(T,N2,Rest).



actions_last_fix(String_to_read, N, [grasp(Helper3)|Rest]):-
    String_to_read =[H|T],
    atom_concat('action(', N, Helper),
    atom_concat(Helper,Helper2,H),
    atom_concat(',grab,',Helper3,Helper2),
    N2 is N+1,
    print(H),
```

```
    actions_last_fix(T,N2,Rest).
actions_last_fix(String_to_read, N, [move(position,Helper3)|Rest]):-
    String_to_read =[H|T],
    atom_concat('action(', N, Helper),
    atom_concat(Helper,Helper2,H),
    atom_concat(',move,',Helper3,Helper2),
    N2 is N+1,
    actions_last_fix(T,N2,Rest).
actions_last_fix(String_to_read, N, [search(Helper3)|Rest]):-
    String_to_read =[H|T],
    atom_concat('action(', N, Helper),
    atom_concat(Helper,Helper2,H),
    atom_concat(',search,',Helper3,Helper2),
    N2 is N+1,
    actions_last_fix(T,N2,Rest).
actions_last_fix(String_to_read, N, [deliver(Helper3)|Rest]):-
    String_to_read =[H|T],
    atom_concat('action(', N, Helper),
    atom_concat(Helper,Helper2,H),
    atom_concat(',drop,',Helper3,Helper2),
    N2 is N+1,
    actions_last_fix(T,N2,Rest).
actions_last_fix([],N,[]).

list_butlast([X|Xs], Ys):-
    list_butlast_prev(Xs,Ys,X).
```

```
list_butlast_prev([],[],_).
list_butlast_prev([X1|Xs], [X0|Ys], X0):-
    list_butlast_prev(Xs,Ys,X1).

remove_chars(X,C,Y,N1,N2):-
    atom_chars(X, Xs),
    select(C, Xs, Ys),
    atom_chars(Z, Ys),
    N3 is N1+6,
    remove_chars(Z,C,Y,N3,N2).
```

VI- Base de datos en Prolog

```
class(drink,comestible,[age=>all,shelf=>shelf1],[],[]),

class(soda,drink,[inv=>0],[],[[id=>s1,[brand=>coke,in=>storage,original_id=>s1],[]]]),

class(beer,drink,[age=>18,inv=>0],[],[[id=>b1,[brand=>heineken,in=>storage,original_id
=>b1],[]]]),

class(bread,comestible,[inv=>0,shelf=>shelf3],[],[[id=>d1,[brand=>bisquits,on_discount,i
n=>storage,original_id=>d1],[]]]),

class(point,top,[],[],[[id=>start,[name=>start],[]],[id=>shelf1,[name=>the shelf of
drinks],[]],[id=>shelf2,[name=>the shelf of food],[]],[id=>shelf3,[name=>the shelf of
bread],[]]]),

class(robot,top,[],[],[[id=>golem,[position=>start],[]]]),

class(states,top,[],[],[]),

class(events,top,[],[],[]),

class(dialog,events,[],[],[[id=>dialog1,[said_by=>assistant,content=>[]],[]]]),
```

class(observation,events,[],[],[]),

class(grasp object,events,[],[],[]),

class(orders,top,[],[],[]),
class(failed_orders,top,[],[],[])

VII- Base de datos en ASP

```
class("drink")
parent("drink",comestible).
property("drink",shelf,shelf1).
property("drink",age,all).

class("soda").
parent("soda",drink).
property("soda",inv,1).
instance("soda",id,s1).
property(s1,brand,coke).
property(s1,original_id,s1).
property(s1,in,storage).

class("beer").
parent("beer",drink).
property("beer",inv,1).
property("beer",age,18).
instance("beer",id,b1).
property(b1,in,storage).
property(b1,original_id,b1).
property(b1,brand,heineken).

class("bread").
parent("bread",comestible).
property("bread",shelf,shelf3).
property("bread",inv,1).
instance("bread",id,d1).
property(d1,in,storage).
property(d1,original_id,d1).
property(d1,on_discount,null).
```

```
property(d1,brand,bisquits).

class("top").
parent("top",none).

class("object").
parent("object",top).

class("point").
parent("point",top).
instance("point",id,shelf2).
property(shelf2,name,"the shelf of food").
instance("point",id,shelf1).
property(shelf1,name,"the shelf of drinks").
instance("point",id,shelf3).
property(shelf3,name,"the shelf of bread").
instance("point",id,start).
property(start,name,"start").

class("robot").
parent("robot",top).
instance("robot",id,golem).
property(golem,position,shelf3).

class("states").
parent("states",top).

class("events").
parent("events",top).

class("dialog").
parent("dialog",events).
instance("dialog",id,dialog1).
property(dialog1,content,[bring(coke,shelf1),bring(heineken,shelf1),bring(noodles,she
lf2),bring(bisquits,shelf3)]).
property(dialog1,[],null).
property(dialog1,said_by,assistant).

class("observation").
parent("observation",events).
```

```
instance("observation",id,observation(shelf1)).
property(observation(shelf1),seen_by,robot).
property(observation(shelf1),observed_objects,[heineken,noodles]).
instance("observation",id,observation(shelf2)).
property(observation(shelf2),seen_by,robot).
property(observation(shelf2),observed_objects,[kellogs]).
instance("observation",id,observation(shelf3)).
property(observation(shelf3),seen_by,robot).
property(observation(shelf3),observed_objects,[bisquits,coke]).

class("grasp object").
parent("grasp object",events).
instance("grasp object",id,grasp(noodles)).
instance("grasp object",id,grasp(coke)).

class("failed_orders").
parent("failed_orders",top).

class("orders").
parent("orders",top).
```