Desarrollo de Agentes Inteligentes para el Control de un Juego de Rol

**Introducción**

El proyecto consiste en el diseño e implementación de un agente inteligente para el control de un personaje jugador de un juego de rol. En esta primera etapa se adopta una representación en Prolog de las creencias que el agente tiene del mundo (y que forman, en parte, el estado interno del agente).

Sobre la creencias que el agente tiene del mundo se usara un algoritmo de búsqueda para encontrar la reliquia que nos consume menos energía, en este caso utilizamos el Algoritmo A\*(estrella) dado en clases, este encuentra el camino optimo (el que se consume menos energía en ir a la meta) de un conjunto de metas.

En el presente informe se detallará la representación del mundo adoptada, como la forma en que ésta se actualiza. Se explicará así también, la utilidad o propósito de recordar la información adicional guardada mediante la percepción del agente

**Representación del mundo y actualización de creencias**

Actualización de mundo.

El agente percibe y sobre esa persecución actualiza sus creencias.

**time/1**

Predicado provisto por la cátedra.

Cuando el agente percibe, se olvida el tiempo que tenía como creencia y guarda el tiempo percibido. Por el momento no se utiliza la creencia del tiempo.

**node/3**

Predicado provisto por la cátedra.

Los nodos son fijos entonces nunca los borramos de la creencia del agente.

**has/2**

Predicado provisto por la cátedra.

has (Entity1, Entity2) indicando que una entidad Entity1 dentro del rango de visión del

agente, posee una entidad Entity2.

**Acerca de la Búsqueda**

Para la búsqueda se utilizó el algoritmo A\*, el cual resulta optimo y completo, debido que se respetaron las condiciones necesarias para completitud y estimabilidad.

Para decidir que metas era conveniente buscar, se pasó por parámetro todas las metas y el algoritmo devuelve cual es la meta en la cual el agente consumiría menos vida en llegar a ella.

La búsqueda se realiza cada vez que deseamos ir a buscar una Meta (cualquier tipo de entidad) .

Predicado **Buscar\_plan\_desplazamiento(+Metas,-Plan,-Destino)**

Este predicado es utilizado como cascara, para llamar al predicado búsqueda con el nodo de la posición actual del agente, con costo cero y camino como una lista vacía y una vez que búsqueda devuelve el camino llama al predicado **acomodarPlan(+Camino,-CaminoConMove)** que a cada nodo “N” del camino lo adorna con move(“N”) y además invierte la lista ya que el camino que devuelve búsqueda esta invertido y me da el destino como primer elemento de la lista.

**busqueda(+Frontera,+Visitados,+Metas,-Camino)**

Es un predicado recursivo:

El caso base se cumple si el primer nodo de la frontera pertenece al conjunto de metas (**+Metas**) pasada por parámetro.

El caso recursivo genera los vecinos del primer nodo de la frontera utilizando **generaVecinos(+Nodo,+Metas,-Vecinos)** , luego a esos **Vecinos** lo agrega a la Fronterera utilizado **agregar(+Frontera,+Visitados,+Vecinos,-FronteraConVecinos,-VisitadosModificados )** y a esa **FrontraConVecinos** la ordenamos por la función F(N) que es costo del camino para alcanzar el Nodo N mas la heurística de N hasta la meta mas cercana y para ello utilizamos **ordenar\_por\_f(+FronteraConVecinos,-FronteraOrdenada) ,** por lo tanto el primer nodo de la **FronteraOrdenada** será el de menor F(N) de todos.

Por ultimo llama recursivamente a **busqueda** con la **FronteraOrdenada** y con la lista de **Visitados** un elemento mas(el primer Nodo que se sacó de la **Frontera** del cual se generaron sus **Vecinos**)

**generarVecinos(+Nodo, +Metas,-Vecinos)**

Este predicado genera todos los nodos que son adyacentes al Nodo, donde cada **NodoAdy** tiene la estructura **NodoAdy**=**nodo(Id,Costo,Camino)** para ello le calcula el costo, que es la heurística del **NodoAdy** mas el costo del **NodoAdy** mas el costo del Nodo. Para agregar el **Camino NodoAdy** es el Camino de **Nodo** agregándole el **Id** de **NodoAdy.**

**agregar(+Frontera,+Visitados,+Vecinos,-FronteraConVecinos,-VisitadosModificados )**

Agrega los vecinos chequeando varias condiciones:

1. Si Vecino no está en Frontera ni en Visitados lo agrega a la Frontera.
2. Si Vecino ya esta en Frontera y el costo F del Vecino es peor del que ya esta en la Frontera no se agrega.
3. Si el Vecino esta en Visitados y el costo F del Vecino es peor del que ya esta en la Visitados no se agrega.
4. Si Vecino ya esta en Frontera y el costo F del Vecino es mejor del que ya esta en la Frontera se elimina el que esta en Frontera y se agrega el Vecino a la Frontera.
5. Si Vecino ya esta en Visitados y el costo F del Vecino es mejor del que ya esta en la Frontera se elimina el que esta en Visitados y se agrega el Vecino a la Frontera.

**Ordenar\_por\_f(+Frontra,-FrontraOrdenada)**

Ordena la Frontera según f que es el costo explicado anteriormente de menor a mayor. Este predicado utiliza el algoritmo de ordenamiento quicksort