PROYECTO
INTELIGENCIA ARTIFICIAL
ETAPA 1

COMISION:

FRANCO LAUTARO CARRANZA

LEANDRO MARA

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).
Utilizando las 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 y la forma en que ésta se actualiza.

Representación del mundo y actualización de creencias

El agente percibe y sobre esa percepción actualiza sus creencias.

time/1

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

node/3

Los nodos son fijos, nunca los eliminamos de la creencia del agente, ósea cuando percibe nuevos nodos los guarda para tener conocimiento de todos los posibles nodos del mapa.

has/2

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

Casos del has:

- Si percibimos que un agente tiene un objeto y en la base de creencias teníamos un at de ese mismo objeto entonces borramos el at e insertamos el has.

- Si percibimos que un agente tiene un objeto y teníamos un has de inn/grave/home de ese mismo objeto entonces eliminamos este has e insertamos el has del agente.

At/2 y AtPos/2

Casos del at y atpos:

- si en la percepción recibimos un nodo N y el agente en la base de creencias tenía un at en N, pero ahora en la percepción NO se recibe el at en N entonces eliminamos el at y el atPos de esa entidad.
  Para chequear esto utilizamos el predicado **actualizarTerreno(Percepcion).**

- Si percibimos la misma entidad que teníamos en la base de creencias, pero ahora en otro nodo/vector entonces actualizamos el at/atPos.

- Si percibimos un at/atPos de un objeto que tenía un agente entonces eliminamos el has de ese agente con el objeto e insertamos el at/atPos.

entity_descr/2

Tenemos solo un caso para este hecho, si percibimos la misma entidad, pero ahora con una descripción distinta actualizamos la descripción.

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

Explicación del código realizado:

Buscar_plan_desplazamiento (+Metas, -Plan, -Destino)

Este predicado es utilizado como cascara, para llamar al predicado **busqueda/4** con el nodo de la posición actual del agente, costo cero y camino como una lista vacía y una vez que **busqueda/4** 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 **busqueda/4** 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 Frontera 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 más la heurística de N hasta la meta más 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** con un elemento más (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 “N”, donde cada **NodoAdy** tiene la estructura **NodoAdy**=**nodo(Id,Costo,Camino)** para ello se calcula el costo(Costo NodoAdy + Costo N + Heurística de NodoAdy) y el **Camino** del **NodoAdy** es el Camino de **N** agregándole el **Id** de **N.**

---

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 está en Frontera y el costo F del Vecino es peor del que ya está en la Frontera no se agrega.

3. Si el Vecino esta en Visitados y el costo F del Vecino es peor del que ya está en la Visitados no se agrega.

4. Si Vecino ya está en Frontera y el costo F del Vecino es mejor del que ya está en la Frontera se elimina el que está en Frontera y se agrega el Vecino a la Frontera.

5. Si Vecino ya está en Visitados y el costo F del Vecino es mejor del que ya está en la Frontera se elimina el que está 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.

---

heuristicas(+VectorNodo,+Metas,-ListasHeuristicas)

Calcula las heurísticas del Nodo con todos los nodos Metas. La heurística elegida es la distancia Euclidiana que dicho predicado que la calcula es dado por la cátedra distance(+Vector1,+Vector2,-Distance).
Cuando calcula todas la heurísticas el resultado lo va insertando ordenadamente en **ListasHeuristicas** por lo tanto el primer elemento del lista será el de menor heurística, ósea el que se va a utilizar para calcularle el costo F al Nodo.
Para insertar ordenadamente utilizamos el predicado **inserta(+Elem,+Lista,-ListaComElem)**

Utilización del predicado
buscar_plan_desplazamiento/3

Desde el archivo principal donde esta codificada las acciones del Agente(agent_template.pl), utilizamos el predicado dinámico **plan/1** para guardar la lista de movimientos que nos devuelve **buscar_plan_desplazamiento/3.**

Por lo tanto, si no existe un plan de acción se llama al predicado **armarListaMetas(+Entidad,-Metas)** que dado un tipo de entidad nos devuelve una lista de nodos donde tienen el identificador y el Vector de donde se encuentran dichas entidades.

Esta lista de Metas es utilizada para llamar a **buscar_plan_desplazamiento/3** y la lista de secuencia de movimiento que devuelve este predicado “L” es guardada en plan(L).

Si existe un plan de acción, se utiliza el primer elemento de la lista del plan y se elimina el plan con la lista entera y se guarda un nuevo plan que contiene la lista sin el primer elemento que se utilizó.

Por ultimo cuando la lista del plan contiene un solo elemento se utiliza dicho elemento para la acción y se elimina el plan. Es este caso, en el próximo ciclo de ejecución se tendría que crear un nuevo plan.