Práctica SID

David Casals Reol
Maria Montalvo Falcón
Juli Sahun Montejano

Index

Introducción y objetivo....3
Metodología....4
Agentes....5
BDIAgent....5
Agente Situado....8
Protocolo de comunicación....8
Explorer....9
Objetivos y Estrategia....9
Arquitectura....9
Conductas y planes....9
Collector....10
Objetivos y Estrategia....10
Arquitectura....10
Conductas y planes....10
Tanker....11
Objetivos y Estrategia....11
Arquitectura....11
Conductas y Planes....11
Behaviours....12
Core....14
Reparto y distribución de tareas....17

Introducción y objetivo

En el siguiente documento se explicarán las diferentes estrategias definidas para el desarrollo de la práctica de sistemas inteligentes distribuidos, especificadas para cada uno de los roles que puede llevar a cabo nuestro agente.

En este documento, se presentarán las estrategias de implementación para los tres roles diferentes en el entorno multiagente con recursos limitados. Estos roles incluyen al explorador o *explorer*, almacenador o *tanker* y recolector o *collector*. Cada uno de estos roles tiene objetivos específicos y se espera que los agentes desempeñen diferentes conductas y realicen tareas para alcanzar dichos objetivos. Aparte, nuestros agentes tienen el objetivo común de recoger entre todos el máximo número de recursos posibles del mapa.

En el documento, además, se describirá la arquitectura general del agente, incluyendo sus componentes principales y su interacción con el entorno.

El fin principal es programar agentes inteligentes que sean capaces de trabajar de manera estable y coordinada, en un entorno con diferentes agentes, pozos, viento, oro, diamantes y gólems.

Como hemos mencionado, hay tres tipos de agente: explorer, recolector y tanker.

1) **Explorer**: recorrer el mapa (o una sección de este) constantemente e informar al resto de con información actualizada. El número de veces que se visitan los nodos ha de tener una varianza pequeña.
2) **Recolector**: recoger el máximo número de recursos que le sean posibles, dados un nivel y un máximo de recursos a cargar, y llevarlos al tanker más cercano siempre que sea posible para optimizar el tiempo de recolección.
3) **Tanker**: almacenar los recursos que los recolectores le dan.

Metodología

Nuestros agentes estarán compuestos de dos agentes realmente, el Agente situado que desempeñará uno de los anteriores roles mencionados y el BDIAgent que se encargará de gestionar la ontología.

El Agente situado es aquel que se encuentra *'físicamente'* en el mapa y sobre él repercuten las acciones. No tratará de razonar mucho, se dedicara a recibir mensajes del entorno, procesarlos y recibir órdenes. En última instancia será este quien recoja los recursos, se mueva de un nodo a otro para explorar el mapa o almacene los recursos dependiendo del rol que ejecute.

El BDIAgent es un agente '*externo'* al entorno, fuera del mapa y su papel es recibir la información del mapa a través del Agente situado, gestionarla y razonar sobre cuál es el siguiente objetivo, movimiento o mensaje entre otros. El BDIAgent basa su comportamiento y razonamiento en una serie de *Beliefs*, *Desires and Intentions,* los cuales serán diferentes dependiendo del rol del Agente situado.

Agentes

BDIAgent

En la inicialización de nuestro agente BDI necesitaremos (1) crear unos *Beliefs* iniciales (2) crear los *Desires* básicos definiendo los *Goals* y (3) asignar *Planes* para los *Goals* y añadirlos a la librería de planes. Estos estarán activos hasta que se den por finalizados los *Desires* (de forma exitosa o no).
Claramente, al ser un agente de tipo BDI su arquitectura es orientada a objetivos es decir, es un agente deliberativo, que razona sobre el entorno que conoce (en este caso a través de la ontología y la información que recaptan los agentes que se mueven por el mapa actualizándola).

Tiene como atributo privado *Goal pingAgentGoal*

initBeliefs():
En esta función declaramos todos los belief de nuestro agente BDI. Los Believes que usaremos son los siguientes:

TransientPredicates:	**TransientBeliefs:**
iAmRegistered	ontology
isExplorerAlive	backPackFreeSpace
isCollectorAlive	mapUpdates
isTankerAlive	goldCapacity
situatedPinged	diamondCapacity
situatedCommanded	rejectedNodes
isFullExplored	currentSituatedPosition
	ontologyHash

La mayoría de los nombres son bastante autoexplicativos, por ejemplo, los believes *isExplorerAlive*, *isCollectorAlive* y *isTankerAlive* nos servirán para saber qué tipo de agente situado nos han asignado.

El Belief onologyHash sirve para filtrar las ontologías de las cuales ya hemos obtenido conocimiento, ya que aprender de una ontología es una tarea costosa.

La Belief mapUpdates contiene una cola de Map con las actualizaciones que nos manda nuestro agente situado y que usamos para actualizar la ontología en cada ciclo del agente BDI

Por último, el Belief rejectedNodes, contiene un diccionario con los nodos que han sido rechazados por el agente situado, y las veces que los hemos intentado comandar estando rechazados. Hemos optado por esta solución, ya que creemos que si el agente situado nos rechaza un nodo, será porque justo habrá un agente en ese momento, y dentro de poco tiempo, lo podremos volver a comandar sin problema

initGoals(): añadimos los *goals* y *plans*
Creamos los goals iniciales necesarios, generamos los goalsTemplates correspondientes para luego poder assignar los planesBodys y añadirlos a la librería de planes.

registerGoal → registerPlanBody
pingAgentGoal → pingAgentPlanBody
conmmandGoal → commandSentPlanBody
y un DefaultPlan para mantener el mailbox vacío → keepMaiklboxEmptyPlan

El BDIAgent tendrá aparte unas funciones para asegurar el correcto funcionamiento en cada ciclo BDI:

- overrideBeliefRevisionStrategy();
  Mira si el mapa está explorado, si lo está, el Belief de isFullExplored lo pondrá a True.

- overrideBeliefRevisionStrategy():
  Hará la actualización del mapa con updateMap de los *updates* encolados, si no hay nada se queda igual.

- overrideOptionGenerationFunction():
  En cada ciclo BDI se decide que *goal* se ha de cumplir. Por ejemplo, si el agente no está registrado se registra en el DF, si no ha contactado con el Agente situado, se le envía el ping, de lo contrario, se le envía el comando correspondiente al tipo de agente que se esté ejecutando en el entorno.

Explorer: Si está vivo y no tiene una orden activa, miramos la siguiente orden que le podemos dar. Generalmente, la orden es: ir a un nodo abierto o al menos visitado siempre que estos sean accesibles. Si el nodo al que "tocaría" ir no es accesible, se genera una especie de ciclo para que el agente no se quede bloqueado. Finalmente, actualizamos su set de goals y el Belief commandSent = True (orden activa) para el ciclo BDI.

Collector: Si está vivo y no ha recibido ninguna orden, pedimos a la ontología el nodo al cual nos tenemos que dirigir. El BDIAgent responderá con un nodo el cual tenga un número mayor que 0 del recurso que queremos recoger. En el caso de que no haya ninguno, se escoge un nodo que contenga recursos aunque sean de un nivel más alto, para ayudar a desbloquear el nodo.

Tanker: Si está vivo y no tiene ninguna orden en curso, pediremos a la ontología que nos indique cuales son los nodos abiertos, y si no hay, nos colocaremos en un nodo que no sea bloqueante y sea seguro para nos colocaremos allí.

- overrideDeliberationFunction():
  Sobreescribimos el filtro que nos retorna secuencialmente los *goals* de un agente.

- overridePlanSelectionStrategy():
  Se encarga de que los planes de cada goal sean los correspondientes (capabilityPlans).

Agente Situado

Protocolo de comunicación

Independientemente del tipo de agente, todos obedecen el mismo protocolo de comunicación. El agente situado, una vez desplegado a la plataforma, está a la espera de un comando de *ping* por parte del agente BDI. Una vez recibido este comando, el agente situado responderá con un *pong* con su información personal (tipo de agente, capacidad de recursos, nivel, etc.) y con el mapa que ve.

El agente situado irá recibiendo comandos por parte del agente BDI y los va a intentar ejecutar. Una vez ejecutado el comando, tanto como si ha podido cumplir su objetivo como si no, mandara una respuesta al agente BDI con:

- su posición actual
- si se ha movido, el mapa que ha descubierto
- si ha recogido recursos, la cantidad recogida

Con esta información el agente BDI podrá actualizar la ontología y preparar el siguiente comando.

Puede darse el caso que nuestro agente situado reciba un mensaje con una ontología, en ese caso, el agente situado se la comunicara al agente BDI, el cual podrá actualizar la ontología.

Hay que tener en cuanta que cada actualización de la ontología (ya sea por un nuevo mapa del agente situado o por una ontología de otro agente) tiene que venir acompañada de una comunicación de la misma para que el agente situado tenga la última versión de la ontología para ser compartida.

Para poder comunicar la ontología a otros agentes, hemos usado la ontología de nuestro compañero **Pol Marcet**, y una clase que nos ha proporcionado para gestionarla.
No obstante, creemos que tanto su ontología como su clase nos eran insuficientes para nuestro objetivo y, por lo tanto, nos hemos visto obligados a ampliarlos.

En la ontología, hemos añadido el atributo *timesVisited* para contabilizar las veces que se visita un nodo.

En la clase *MapaModel* hemos añadido funciones para realizar *queries* sobre la ontología, como por ejemplo `getResourceNodes` que nos devuelve los nodos con recursos del mapa.

Explorer

Objetivos y Estrategia

El objetivo del explorador es descubrir y recaptar información sobre el entorno de manera que mantenga de forma más o menos uniforme el número de veces que se visita cada nodo, es decir, que no se visite un nodo diez veces mientras otros se visita sólo una (obviamente, siempre que sea posible). Por un lado, el objetivo es explorar el máximo número de nodos posibles, pero, por otro lado, se requiere mantener la información actualizada.

Para lograr esto, el explorador utilizará una estrategia basada en la exploración activa. El agente realizará movimientos en el entorno desplazándose al nodo que menos veces haya visitado siempre que sea posible (lo que requiere que sea accesible), prestando atención en las áreas que aún no han sido exploradas (inicialmente, los nodos están abiertos y no se cierran hasta que se han visitado).

Detectará la presencia de recursos o información en el entorno y actualizará su conocimiento sobre los nodos comunicándoselo al BDIAgent a través de mensajes. También queremos que dicha exploración sea hecha de la forma más equitativa posible para minimizar la varianza entre el número de veces que ha sido visitado cada nodo.

Como atributos privados tenemos el mapa, el maprepresentation, nodesRejectats, missatges, behaviourStatus, behaviours, aurguments, onntology, ontologySharedId
y el identificador del BDIAgent que será público.

Arquitectura

En este caso, la arquitectura de nuestro agente es de tipo reactiva con estado interno. El agente no razona sobre el mapa ya que recibe las órdenes de nuestro agente BDI, pero sí que contiene información y por tanto un estado más o menos actual del entorno que va modificando y actualizando al BDI.

Conductas y planes

Como explorador queremos transmitir información actualizada a nuestro entorno lo antes posible por lo que nos centramos en mantener el mailbox vacío y gracias a KeepMailboxEmptyPlan tenemos como gestionar el buzón de mensajes. De otro modo, CommandSentPlanBody nos permite saber que nos está pidiendo exactamente el entorno.

Tenemos un OneShotBehaviour que registra al Agente situado y añade un CyclicBehaviour compuesto llamado Listener que en función de los mensajes que reciba hará una acción u otra mediante matching. Estos sub-behaviours se irán añadiendo según las órdenes del BDIAgent.

Collector

Objetivos y Estrategia

El objetivo del recolector es maximizar la recolección de recursos individuales y globales. El recolector utilizará una estrategia basada en la toma de decisiones óptima. El BDIAgent evaluará la disponibilidad de recursos, el espacio libre en la mochila y el nivel de los recursos para tomar la siguiente acción a realizar.
El recolector buscará maximizar su desempeño individual, considerando la cantidad y calidad de los recursos recolectados, así como la competencia con otros recolectores en el entorno. A lo largo del camino de un nodo X a un nodo Y además, el agente recolector enviará la información del mapa a través del método mapUpdate(). Además, siempre que recoja los recursos, actualizará la información del mapa, poniendo como valor del número de recursos el máximo entre lo que había menos lo recolectado y 0.
En el caso de que no haya ningún recurso con nivel para que lo pueda recoger, se dirigirá a un nodo abierto, para explorar el mapa. En caso de que lo queden nodos abiertos, o sea que el mapa está completo, se desplaza al lado de un nodo con recurso de nivel mayor para ayudar a desbloquearlo si algún agente se acerca.
Es importante tener en cuenta que al tener un espacio limitado de carga, interesa poder vaciar la mochila regularmente para que no se quede llena y nos impida recoger algún recurso de forma completa.

Arquitectura

Como en el caso anterior, la arquitectura de nuestro agente collector es de tipo reactiva con estado interno. El agente tampoco razona sobre el mapa, ya que recibe las órdenes de nuestro agente BDI sobre a dónde debe moverse y una vez se encuentra allí, recoger el recurso correspondiente. De igual forma, contiene información sobre el entorno y, por tanto, un estado más o menos actual que va modificando a medida que se deseplaza o modifica y lo actualiza al BDI.

Conductas y planes

El agente recolector tiene como conductas y planes: el OneShotBehaviour RegistertoDF, el MessageMapper y un TickerBehaviour llamado BackpackEmptier. Los iniciamos a través de startMyBehaviour para asegurar que todo está correctamente en los contenedores correspondientes y podemos movernos por el entorno.
El MessageMapper es un OneShotBehaviour que añade un CyclicBehaviour compuesto llamado Listener que en función de los mensajes que reciba hará una acción u otra mediante matching. Estos sub-behaviours se irán ejecutando de manera que se cumpla su rol.

Los planes respecto al Mailbox y los comandos del BDIAgent son los mismos que los mencionados en el explorer; pero las conductas son distintas ahora. En este caso, nuestro agente está interesado en ir a un nodo donde sepamos que se encuentra algun recurso que podamos recolectar. También tenemos una capacidad máxima de cargar recursos, por lo que es necesario vaciar nuestra mochila y darle a un tanker todo aquello recogido.

Tanker

Objetivos y Estrategia

El objetivo del agente de almacenamiento es almacenar los recursos. Este agente recibirá órdenes inferidas a partir de la información para colocarse en nodos que no sean bloqueantes y que sean seguros.
En sí, no nos dedicaremos a perseguir a los recolectores para que nos den sus recursos, sino que se podría decir que esperaremos a que se acerquen.

Arquitectura

Finalmente, en este último caso, la arquitectura de nuestro agente tanker es de tipo reactiva con estado interno. El agente tampoco razona sobre el mapa ya que recibe las órdenes de nuestro agente BDI sobre a dónde debe colocarse (al lado de un nodo con recurso) y una vez se encuentra allí, esperar a que un collector pase a recoger recursos y se los deje. De igual forma, contiene información sobre el entorno y por tanto un estado más o menos actual que va modificando a medida que se desplaza de un lugar a otro cuando le indica el agente BDI.

Conductas y Planes

Los planes del tanker en relación al Mailbox y los comandos son iguales a los otros agentes, solo que su conducta es muy distinta.
El tanker tiene como objetivo almacenar los recursos que los recolectores van recogiendo por el mapa por lo que hemos optado por un comportamiento más pasivo en el cual el tanker siempre que no se encuentre en un nodo de paso o cuello de botella y esté penalizando al grupo por estar en un nodo inaccesible, se quedarà quieto de manera que si cualquier recolector pasa cargando recursos, será él quien se acerque.

Behaviours

MessageMapper:
MessageMapper genera un HashMap de potenciales acciones con un identificador y el plan de accion y añade este HashMap como un Listener Behaviour (Cyclic Behaviour).
Estas poteciales acciones son tienen como identificadores:
-**position**:
Si la posicion del JSONObject es nulla o es un rejectedNode, se indica con un INFORM o un REJECT_PROPOSAL respectivamente.
Sino se envía un ACCEPT_PROPOSAL, se registra un WalkTo, se crea un HashMap de posibles respuestas a los distintos casos (SUCCES|FINISH|ERROR) y se crea un nuevo behaviour Composer action, que se añadirà. Composer es un sequential behaviour que crea 2 subbehaviours, maini calbacks. main es el WalkTo que se ejecutará primero y al finalizar irá callbacks que es el HashMap con las posibles respuestas en función del Status (actualizado por el WalkTo)

-**collect**:
Se registra un PickItem Behaviour, crea un HashMap para los distintos finales (SUCCES, ERROR) y crea un Composer Behaviour llamado action y lo añade a la lista de behaviours.

-**map**:
Enviamos la ontología del agente mediante el TickerBehaviour OntologySharer al resto de agentes del entorno.

-**ontology**:
Obtenemos los datos del JSONObject a través de la transformación del mensaje a JSON con la clase Listener. Una vez lo tenemos, enviamos un mensaje al máster con la información recién obtenida.

-**ping**:
Se da cuando se ejecuta el plan de SITUATED_PINGED.
Obtenemos el identificador del sender, y el actualizo atributo publico master a este nuevo. Genero un mensaje con mi posición, mi tipo de agente, mi mapa y mi capacidad de recursos. Envío el mensaje a mi máster, con el fin de empezar la conexión.

-**noOp**:
Comando usado principalmente por el tanker. Dicho comando especifica que la posición o nodo en el que se encuentra el agente no es un nodo de paso, por ello, no está afectando negativamente al resto de agentes.

Listener:
El Listener es un CyclicBehavour que contiene estas 6 posibles acciones. Estará bloqueado hasta que le llegue un mensaje. Cuando llega un mensaje, lo que hará es comparar la key del mensaje ya mapeado con una de estas acciones que puede hacer nuestro Agente situado. Si hay matching se acepta la parte derecha, que puede ser nada o un mensaje de respuesta. (accept.(Consumer<String>) llevándose a cabo si se puede) con todos los subbehaviours que se puedan crear en función de la key.

Composer:
Composer es un SequentialBehaviour que puede recibir una lista de Behaviours un 2 behaviours y los ejecuta secuencialmente.

ConditionalBehaviour:
Es un behaviour que dado el estado de un behaviour ejecuta el plan de ejecución asociado a ese Status (son JSONObjects) con la información del estado del behaviour indicado.

WalkTo:
Es un SimpleBehaviour. Si la posición actual es la posición target, ha acabado. Sino añadiremos el nodo donde estamos al mapa .Diremos que està explorado cuando no haya OpenNodes o tampoco Unexplorats, llavors fullExplored = True. Cerramos el nodo donde estamos. Si el nodo resulta ser inalcanzable se retornara True.

OntologySharer:
Es un TickerBehaviour con un periodo de 100 ciclos que crea un mensaje SHARE-ONTO con la ontología y se lo envía al resto de agentes.
También tiene una función que actualiza la ontología propia utilizada cuando somos nosotros quienes queremos actualizarnos.

PickItem:
Es un SimpleBehaviour que mira si en el nodo en currentPosition se puede recoger recursos. En caso afirmativo actualiza el status del agente a 0 (SUCCES,) sino 1 (ERROR) y acaba.

BackUpEmptier:
Para cada tanker que tenemos miramos si el Agente situado (Collector) puede hacerle un drop su contenido a dicho tanker. En caso de poder hacerse, se enviará un mensaje INFORM al BDI

MessageSender:
Se encarga de enviar los mensajes con el contenido bien definido.

Core

Decidimos crear una serie de clases que pudieran ser útiles para las clases del proyecto. Entre ellas están:

Constants:

En esta clase hay un conjunto de Strings que determinan constantes. Entre ellas están por ejemplo: `SITUATED_PINGED` que indican si el Agente situado está vivo o `IS_FULL_EXPLORED` si el mapa está totalmente explorado, etc.

Map:

Esta clase está implementada como una clase serializable. El objetivo de esta clase es facilitar la comunicación del mapa entre el agente situado y el agente BDI, ya que la clase MapRepresentation nos era insuficiente. Entre sus elementos encontramos:

```
private HashMap<String, Node> nodes;
//represena una lista de los nodos, indexados por su identificador

public Map()
//inizializa el HashMap a uno vacío

public Set<String> keySet()
public Node get(String key)
public Boolean has(String key)
public void put(String key, Node value)
public void clear()
//metodos para insertar u obtener elementos

public void visit(String nodeId)
//registra una visita al nodo indentificado por el parametro

public String stringifyNodes()
//convierte la lista de nodos a JSON para luego convertirlo a String
```

MapaModel:

Esta clase instancia Model de Jena para almacenar y manipular datos RDF. Tiene un constructor el cual inicializa el modelo. MapaModel es la clase que nos ha servido para contener información sobre los nodos y los agentes en el mapa. Entre otras cosas, los métodos de la clase MapaModel permiten agregar y actualizar información sobre el entorno. Algunos de los atributos principales de la clase MapaModel incluyen:

```
public class NodeInfo;
//representa un nodo con su informacion

public class AgentInfo;
//representa un agente con sus propiedades

public Model model;
//representa el modelo de la ontologia

public void addNode(String id, NodeType type);
//añade un nodo con el estado (abierto o cerrado) que le indiquemos

public void addAdjancency(String node1id, String node2id);
//añade una arista entre los nodos indicados

public Set<String> getOpenNodes();
//devuelve los nodos abiertos de la ontologia

public void learnFromOtherOntology(MapaModel otherModel);
//actualiza la ontologia con la información mas nueva que pueda contener
la nueva
```

Message:

Esta clase la hemos utilizado para representar un mensaje en el contexto de JADE. El objetivo de esta clase es hacer simplificar el código y hacerlo más legible. Contiene los siguientes atributos

```
public String sender;
public String ontology;
public String protocol;
public String content;
public String receiver;
public int performative;

public Message(ACLMessage msg, String receiver);
```

Node:

La clase Node representa un nodo del entorno. Algunos de los atributos y metodos a destacar son los siguientes:

```
private Status status;
private String id;
private Integer timeVisited;
private HashSet<String> neighbors;
private List<Couple<String, Integer>> observations;
//atributos para representar el nodo

public Node(JSONObject json);
//crea un nodo a partir de uno en formato JSON

public void visit()
//visita el nodo

public void updateResource(String resource, Integer amount)
//actualiza la cantidad de recurso que contiene ese nodo

public void mergeObs(List<Couple<Observation, Integer>> observations)
//fusiona las observaciones del nodo con las enviadas como parametro

public JSONObject toJson()
//pasa a formato JSON el nodo para poder ser comunicado
```

Utils:

Esta clase la hemos utilizado para definir varios métodos estáticos que se han utilizado a lo largo del proyecto.

```
public static String uuid()
//genera un uuid

public static String registerBehaviour(Agent a, Behaviour b, String id)
//registra el behaviour b para el agente a con el id

public static void finishBehaviour(Agent a, String id, Integer code)
//marca al behaviour como finalizado con el codigo indicado

public static void sendMessage(Agent a, int performative, String
   content, String to)
//manda un mensaje
```

```
public static void updateMap(Map patchUpdate, MapaModel ontology,
   HashMap<String, Integer> rejectedNodes)
//actualiza la ontologia con el mapa patchUpdate

public static HashSet<String> getTankers(Agent a)
// obtiene los agentes tankers del DF

public static String findAgent(Agent a, String name)
//busca a un agente llamado como el parametro

public static Message messageMiddleware(Agent a, ACLMessage msg)
//hace de middleware para evitar procesar menssaje de otros agentes

public static JSONObject getBackPackFreeSpace(SituatedAgent03 agent)
//obtiene el espacio sobrante del agente
```

Reparto y distribución de tareas

Juli Sahun: Ha realizado la mayor parte de la implementación de ambos agentes, sus comportamientos y sus protocolos de comunicación.

Maria Montalvo: Ha colaborado en la implementación de la práctica. Ha ayudado a tomar decisiones sobre objetivos y planes de los agentes. Ha redactado casi en su totalidad el documento de la entrega.

David Casals: No ha colaborado en la implementación de la práctica. Ha intentado redactar parte del documento de la entrega, pero al no tener el suficiente conocimiento del proyecto, numerosas partes han sido reescritas.