Práctica Dedale SID

(Cuatrimestre de Primavera)

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

19 de Junio de 2022

Jia Long Ji Qiu
Victor Teixidó
Jiabo Wang
You Wu

Índice

Características principales del agente BDI y el agente situado en el entorno **2**

- Inicialización del agente BDI 2
- Inicialización del agente situado en el entorno 3
- Proceso de sincronización (SynchroniseBehaviour, SynchronisePlanBody) 4
- Redireccionamiento de mensajes (RedirectMessageBehaviour) 4
- Percepción del entorno (SensorActuatorBehaviour) 4
- Razonamiento (ReasonPlanBody) 5

Estrategia diseñada para cada rol: **6**

- Explorador (explorer) 6
- Recolector (collector) 10
- Almacenamiento (tanker) 13

Organización y distribución de las tareas **16**

Características principales del agente BDI y el agente situado en el entorno

Tanto el agente BDI como el agente situado son principalmente agentes reactivos, pero también cumplen la función de agente activo (para iniciar la comunicación, enviar percepciones…), como se verá más adelante.

Inicialización del agente BDI

El agente BDI puede recibir hasta 1 parámetro de entrada, el cual corresponde con el tipo de servicio del agente situado, definido por defecto como "*EnvironmentAgentJTWW*" y utilizado para la búsqueda del agente situado. Se ha implementado para facilitar la incorporación de múltiples agentes a partir de las mismas clases.

Consta de dos *plan bodies*, *SynchronisePlanBody* y *ReasonPlanBody*, asociados a *sequential goals* que consisten en la sincronización con el agente situado y el razonamiento de la información que recibe de dicho agente, de manera que el agente BDI no puede empezar a razonar hasta haberse sincronizado correctamente con el agente situado.

La base de conocimiento consta de los siguientes creencias:
Utilizados para el proceso de sincronización:

- **SynchroniseStatus**: representa el estado en el que se encuentra el proceso de sincronización
- **EnvironmentAgentAID**: representa el AID del agente situado
- **EnvironmentServiceType**: representa el *service type* del agente situado

Utilizados para identificar las características del agente situado

- **EntityType**: representa el rol del agente situado
- **TreasureType**: representa el tipo de tesoro compatible con el agente situado
- **BackpackMaxSpace**: representa la capacidad máxima del agente situado
- **ExpertiseValue**: representa el nivel de habilidad de *lockpicking*
- **BackpackFreeSpace**: representa la capacidad restante del agente

Utilizados para guardar la información sobre el entorno

- **ForeignAgents**: representa los AID de los agentes situados de la plataforma, clasificados según el rol
- **AgentPositions**: representa la última posición en la que se ha visto un agente situado
- **OccupiedPositions**: representa información más precisa sobre las posiciones de los agentes en las cercanías (dependiendo del rango de comunicación)
- **ResourcesInformation**: representa la información sobre los recursos situados en el mapa
- BroadcastMessage: representa un mensaje ACL que tiene a todos los agentes situados como receptores
- Map: representa el mapa interno del sistema
- **MapRepresentation**: representa el *MapRepresentation* utilizado para compartir la topología entre agentes situados
- **OpenNodes**: representa los nodos sin visitar

- **ClosedNodes**: representa los nodos visitados

Utilizados para el razonamiento relacionado al rol de explorador

- **FirstTick**: utilizado para ejecutar ciertas instrucciones en la primera recepción de información del agente situado
- Exploring: utilizado para determinar qué estrategia debe seguir el agente situado
- Ticks: utilizado como contador para reiniciar las posiciones ocupadas en las cercanías
- **TimesVisited**: utilizado para mantener constancia sobre cuántas veces se ha visitado un nodo
- Attempts: utilizado para mantener constancia sobre las veces que se ha intentado ir hacia un mismo nodo
- SelectedNode: utilizado para guardar el último nodo al que se haya querido visitar
- **PreviousLessVisitedNode**: utilizado para guardar el último nodo seleccionado con menos visitas
- **GolemPositions**: utilizado para mantener constancia de dónde hay posiciones de gólems
- **WindPositions**: utilizado para mantener constancia de dónde hay posiciones de viento

Inicialización del agente situado en el entorno

El agente situado en el entorno puede recibir hasta 1 parámetro de entrada, el cual corresponde con el tipo de servicio con el cual se debe registrar, definido por defecto como "*EnvironmentAgentJTWW*". Se ha implementado para facilitar la incorporación de múltiples agentes a partir de las mismas clases.

Antes de iniciar los *behaviours*, el agente situado debe comprobar sus características y registrarse en el DF según estas. Además del servicio mencionado previamente, debe añadir más servicios según el rol que tenga: "agentExplo" si es explorador, "agentCollect" si es recolector y "agentTanker" si es almacenador. Además, los agentes recolectores y almacenadores deberán añadir los servicios "agentGold" y/o "agentDiamond" según la compatibilidad que tengan con el tipo de recurso. También es necesario que se suscriba al DF de los estos tipos de servicio con tal de poder recibir dinámicamente notificaciones sobre el registro del resto de agentes.

En cuanto a los *behaviours*, consta de dos *behaviours* secuenciales, siendo el primero el proceso de sincronización y el segundo un *sub-behaviour* paralelo consistente en el redireccionamiento de mensajes ajenos al agente BDI y la funcionalidad de sensor-actuador, de manera que el agente situado no podrá interactuar con el entorno hasta que se haya sincronizado correctamente con el agente BDI.

Proceso de sincronización (SynchroniseBehaviour, SynchronisePlanBody)

La comunicación entre estos agentes se establece mediante un protocolo de sincronización básico:

- El agente BDI, en el estado “0”, busca en el Directory Facilitator un agente que coincida con el servicio con el que se haya registrado el agente situado en el entorno. Una vez encontrado, se guarda su AID para facilitar futuras conversaciones y le envía un mensaje de sincronización junto a otros mensajes necesarios que piden ciertas características del agente situado, y pasa al estado “1”.
- El agente situado, en el estado “0”, tras recibir dicho mensaje, se guarda el AID del agente BDI (también para facilitar futuras comunicaciones), y pasa al estado “1”.
- El agente situado, en el estado “1”, irá respondiendo las peticiones del agente BDI.
- El agente BDI, en el estado “1”, irá actualizando los datos recibidos del agente situado hasta que todos hayan sido establecidos.
- Finalmente, cuando el agente BDI haya terminado de establecer todos los datos necesarios, enviará al agente situado una señal indicando que el proceso de sincronización se ha terminado, a la cual el agente situado también responderá con otra señal. Recibir la señal implica la finalización del *behaviour* del agente situado y el *plan body* del agente BDI.

NOTA: Es importante que los servicios tengan nombres únicos para evitar una comunicación equívoca.

Redireccionamiento de mensajes (RedirectMessageBehaviour)

El *behaviour* del redireccionamiento de mensajes tiene como objetivo tratar al agente situado como un túnel de comunicación entre el resto de agentes situados y el agente BDI.

Debido a que el agente situado carece de razonamiento, este es incapaz de tratar los mensajes recibidos de otros agentes situados y por lo tanto necesita enviarlos al agente BDI con el que está asociado. Para ello, “empaqueta” un mensaje dentro de otro, con tal de evitar pérdidas de información del mensaje recibido.

Finalmente, es el agente BDI quien se encarga de tratar los mensajes recibidos y decidir en consecuencia a ello.

Percepción del entorno (SensorActuatorBehaviour)

El mecanismo de sensor-actuador del agente situado consiste en una alternancia de dichos estados marcados por un *TickerBehaviour* que ejecuta un tick cada 10 ms. Este ticker permite al agente BDI tener un margen para hacer el razonamiento y poder enviar todas las acciones que el agente situado debería ejecutar.

Ya que la idea es que el agente situado comparta información sobre posiciones, recursos y topología, dicha información se debería enviar en la ejecución de cada movimiento, por lo que debería haber una cierta sincronización dentro del sensor-actuador. Al utilizar un *CyclicBehaviour*, por ejemplo, el movimiento suele ser tan rápido que la información enviada puede estar desactualizada. Aunque el entorno sea asíncrono, esta decisión no afecta al tratamiento del resto de mensajes, ya que el redireccionamiento de mensajes y el razonador sí que funcionan de manera cíclica.

Razonamiento (ReasonPlanBody)

El razonador del agente BDI es, en esencia, un receptor de mensajes. Para cada mensaje recibido, hace un tratamiento distinto dependiendo de su contenido (ontología, objeto contenido, información contenida…).

Cuando el mensaje contiene un mensaje (empaquetado por el agente situado) y, según la ontología del mensaje, este puede tratarse de:

- **FIPA-Agent-Management**: es una notificación enviada por el agente DF cuando un agente se registra a un servicio suscrito. El tratamiento consiste en guardar en el belief “ForeignAgents” el AID del agente según el tipo con el que esté registrado, además de añadir nuevos receptores al belief “BroadcastMessage”.
- **agentPositions**: es información sobre las posiciones de los agentes percibida por el emisor del mensaje contenido. Según el momento en el que se haya percibido una cierta posición es más reciente o no, el razonador puede decidir actualizar o no la posición de un cierto agente en el belief “AgentPositions”. De esta manera, se podrá mantener la máxima consistencia posible en la información enviada.
- **resourceInformation**: es información sobre los recursos observados en el mapa. Al igual que las posiciones de los agentes, el razonador también puede decidir si actualizar o no la información que tiene en el belief “ResourcesInformation” según si es información más reciente o no. El belief “OccupiedPositions” también se actualiza pero únicamente con la posición del agente emisor, de esta manera, dicho belief solo guardará posiciones cercanas (más precisa).
- **mapTopology**: es la información sobre el mapa descubierto por el emisor del mensaje contenido. Se hace un *merge* con el belief “MapRepresentation” y se actualiza el mapa interno “Map” junto a los nodos abiertos “OpenNodes” y cerrados “ClosedNodes” según el contenido del mapa recibido.

Cuando el mensaje consiste en la observación por parte del agente situado, el razonamiento sigue la estrategia diseñada para cada rol, los cuales se explicarán en el siguiente apartado.

Estrategia diseñada para cada rol:

Explorador (explorer)

El explorador persigue dos objetivos: el objetivo principal (llamado plan A) consiste en visitar todos los nodos del mapa, mientras que el secundario (llamado plan B) persigue el objetivo individual del explorador, especificado en el enunciado de la práctica. El pseudocódigo del razonamiento sobre cómo debería actuar el explorador es el siguiente:

```
TratarObservacionExplorador:
	Inicialización de beliefs
	++tick
	IF tick == 50:
		posicionesGolem.clear()
		posicionesAgentes.clear()
		tick = 0
	// Ejecutar plan A
	IF explorando || primer tick || nodos abiertos > 0:
		++visitas(nodoActual)
		siguienteNodo = hacerDFS()
		// Se ha encontrado un movimiento válido
		IF siguienteNodo != null:
			Enviar la propuesta de movimiento al agente situado
			IF nodoSel == null:
				nodoSel = nextNode
			// El nodo seleccionado ya fue seleccionado en el tick
			anterior: el agente situado no consigue moverse
			ELSE IF nodoSel == nextNode:
				++intentos
			ELSE:
				nodoSel = nextNode
				intentos = 0
			// Posiblemente hay un golem en el siguienteNodo
			IF intentos == 10:
				posicionesGolem.add(nextNode)
			// El agente está atascado, pasar al plan B
			ELSE IF intentos == 50:
				explorando = false
				attempts = 0
		// No se ha encontrado ningún movimiento válido
		ELSE:
			++intentos
			IF intentos == 50:
				// Pasar a plan B
				explorando = false
				attempts = 0
```

```
// Ejecutar plan B
ELSE:
	siguienteNodo = null
	// El agente se encuentra en un nodo abierto
	IF nodosAbiertos.contains(nodoActual):
		siguienteNodo = hacerDFS()
	// No se ha encontrado un nodo válido: primero mirar si hay
	nodos abiertos en una adyacencia
	IF siguienteNodo == null:
		camino = {}
		FOR adyacencia : adyacencias:
			IF nodosAbiertos.contains(adyacencia) &&
			hayCamino(nodoActual, adyacencia, camino):
				siguienteNodo = adyacencia
				BREAK
			camino = {}
	// No hay nodos abiertos adyacentes: buscar el nodo menos
	visitado
	IF siguienteNodo == null:
		Ordenar los nodos según las visitas
		camino = {}
		FOR nodo : nodosOrdenados:
			IF hayCamino(nodoActual, nodo, camino):
				siguienteNodo = nodo
				BREAK
			camino = {}
	// Siempre habrá un nodo válido a no ser que el agente esté
	atrapado por otros agentes
	IF siguienteNodo != null:
		IF nodoAnterior = null:
			nodoAnterior = siguienteNodo
		// Se ha hecho un cambio en el nodo seleccionado
		ELSE IF nodoAnterior != siguienteNodo:
			visitas(nodoAnterior) += 5
		// Es posible que el siguiente nodo no sea el adyacente
		IF camino.size() > 0:
			siguienteNodo = siguienteNodoCamino(nodoActual,
			siguienteNodo, camino)
		visitas(nodoActual)++
		Enviar propuesta de movimiento al agente situado
		Utilizar la misma estrategia de detección de golems del
		plan A
Actualizar belief de posiciones de agentes
Actualizar belief de información de recursos
Actualizar belief de topología (implícito en el DFS)
Enviar información al agente situado para su redireccionamiento
```

En el tratamiento de una observación del agente, antes de iniciar una búsqueda del siguiente nodo, se incrementa el Belief “Tick” de manera que cada vez que llegue a 25 se reinician las posibles posiciones ocupadas por golems y agentes. Esto se debe a que, al estar en un entorno cambiante y con limitaciones en el rango de comunicación, es necesario refrescar de vez en cuando estas posiciones para evitar información falsa que se puede producir, por ejemplo, cuando hay un bloqueo y el agente no ha recibido información actualizada. Sin embargo, este reinicio puede provocar que un explorador, tras alejarse después de detectar un obstáculo, quiera volver al mismo nodo seleccionado previamente pasados esos ticks, por lo que es importante que sea un número considerable de ticks para que, al menos, no se hagan recorridos en bucle en un conjunto pequeño de nodos. Los ticks se deben adaptar al tamaño del mapa.

En el plan A el razonador se dedica a buscar nodos sin explorar mediante un algoritmo de DFS. El algoritmo de DFS implementado es el convencional pero con una pequeña optimización: cuando todas las adyacencias están visitadas, se necesita volver a un nodo abierto encontrado anteriormente, dicho nodo abierto se priorizará para la siguiente iteración (poniéndolo en la primera posición de la lista de nodos abiertos). De esta manera, una vez se cambie la selección, se mantendrá para la siguiente invocación. Esta modificación se ha hecho debido a que, por el mismo motivo que antes, el agente podría estar cambiando la selección constantemente entre dos nodos abiertos debido a que se puede dar el caso en el que una vez se aleja de uno, se reinician las posiciones ocupadas y en la siguiente invocación del DFS se vuelva a seleccionar el nodo del cual se quería alejar debido a una ocupación en dicho nodo.

Cuando se encuentra un nodo válido, este se envía al agente situado y se incrementa las veces visitadas del nodo actual. Esto será de ayuda para el plan B.

Cabe destacar también la estrategia que se utiliza para evitar colisiones con golems u otros agentes sin movimiento o que no se comunican. Para la detección de estas colisiones, se guarda en un Belief “SelectedNode” la selección anterior del DFS. Siempre que el siguiente nodo sea igual al nodo anterior seleccionado, querrá decir que el agente no se ha podido mover de su sitio. Por lo tanto, se incrementa el belief “Attempts” de manera que cuando llegue a 10 se considerará que hay un gólem en el siguiente nodo, y si se da el caso en el que se queda quieto de manera prolongada (hasta 50 intentos), posiblemente porque no quedan otros nodos abiertos por visitar, se cambiará al plan B.

Además, si el DFS del plan A no ha encontrado ningún nodo al que se pueda mover, se incrementarán los intentos como si se tratara también de una colisión. Una vez llegue a 50, se cambiará al plan B.

En el plan B, el razonador intenta equilibrar las visitas a cada nodo, de acuerdo al objetivo individual de un explorador. Para cumplir este objetivo será de gran ayuda el belief de “TimesVisited”, ya que servirá para encontrar el nodo válido menos visitado posible. Para ello, la lista de nodos en “TimesVisited” se ordena según las visitas hechas, y se itera sobre cada nodo de menos visitado a más visitado hasta encontrar un nodo válido. De esta manera, el explorador podrá evitar estar demasiado tiempo en una cierta zona del mapa, ya que tendrá que moverse siempre hacia la zona que menos haya visitado en cada tick.

Sin embargo, en este plan no debemos olvidarnos de que puede ser posible que aún queden nodos sin visitar, por lo que es importante darles prioridad cada vez que se encuentra uno. Cuando en el plan B el agente se encuentra en un nodo abierto, ejecutará DFS para añadir sus posibles adyacencias. Si se encuentra en un nodo cerrado y tiene nodos abiertos adyacentes, les dará prioridad siempre y cuando pueda ir hacia ellos. Si no se han encontrado nodos abiertos, entonces se seleccionará el nodo menos visitado.

Una diferencia importante respecto al plan A es que en este plan los nodos seleccionados rara vez son adyacentes, tal y como se seleccionan mediante el algoritmo de DFS. Aún así, se puede lograr un comportamiento parecido a la hora de decidir la selección de nodos y la prioridad que se les debe dar. Para ello se utiliza el belief “PreviousLessVisitedNode” que, tal y como indica su nombre, guarda el nodo anterior menos visitado seleccionado. En el pseudocódigo se puede observar que cuando el siguiente nodo (antes de aplicar la función que retorna el siguiente nodo del camino) es distinto al anterior, se incrementan por 5 las visitas al nodo anterior. Esto, aunque a priori parezca estar en contradicción con el cumplimiento del objetivo individual (porque se están sumando visitas que realmente no se han hecho), a la larga permitirá lograr mejores resultados ya que se trata de un penalizador que evita el cambio sucesivo entre dos nodos (como pasaba con un DFS sin cambio de prioridad, comentado en el plan A) ya que, una vez más, debido al reinicio de las posiciones posiblemente ocupadas, se pueden crear bucles en los movimientos producidos por el agente. Utilizar dicha penalización permitirá evitar volver a escoger un mismo nodo una vez se aleje de un nodo bloqueado y permitirá al agente situado desplazarse hacia otros nodos sin provocar estos bucles, los cuales empeoran el resultado del objetivo individual ya que pueden llegar a incrementar de manera considerable la desviación estándar de visitas.

Para evitar colisiones, se implementa la misma estrategia utilizada en el plan A.

Finalmente, una vez finalizado el tratamiento de la observación, se envían mensajes con mensajes “Broadcast” que contienen información sobre las posiciones, los recursos y la topología que el explorador ha podido ir recopilando.

NOTA: en el código hay un fragmento comentado que sirve para evitar visitar nodos de viento. Esto se ha implementado de manera opcional para evitar mapas con información insegura (que tengan, por ejemplo, nodos con pozos como nodos abiertos) por otros grupos. Al final no se ha optado por ello por si se utilizan mapas con casos especiales como el de la figura 1. La decisión final ha sido que el explorador no trate los mapas recibidos de otros agentes, ya que es éste quien se dedica a explorar el mapa, pero sí lo comparte.

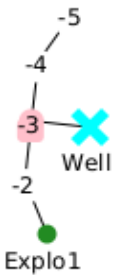

Figura 1. Caso especial en el que se debe atravesar sí o sí un nodo con viento para llegar a otro fragmento del mapa.

Recolector (collector)

El principal objetivo de los agentes recolectores es la maximización de la recogida de minerales durante la ejecución de la partida. Cabe destacar que los minerales que un agente recolector puede recoger son de un único tipo y es por esto que, de cara a los demás agentes, al registrar a dicho agente le designaremos como “agentGold” o “agentDiamond”. De cara al propio BDI que acompañará al agente, tendremos un *belief* donde almacenaremos el tipo de tesoro que puede recoger el recolector, adicionalmente también guardaremos como *beliefs* tanto la habilidad de *lockPicking* como la capacidad de almacenaje de su mochila.

Teniendo en cuenta estos *beliefs* que tendrá el agente BDI, vamos a ver cómo funcionará la comunicación entre ambos y por ende, el comportamiento del agente situado. Tal y como hemos comentado anteriormente, el agente BDI y el situado se irán turnando las comunicaciones de tal manera que el agente situado actuará de sensor enviando las observaciones que perciba y, cuando el BDI haya tomado las decisiones, hará de actuador en el propio mapa. A todo esto, en cualquier momento el agente BDI será capaz de recibir e interpretar mensajes de otros agentes para, por ejemplo, poder ir actualizando la información percibida en el mapa.

El agente BDI indicará mediante distintas ontologías que acciones deberá hacer el agente situado, además, le enviará la información pertinente para la correcta actuación. Vamos a comentar ahora las distintas acciones que podrá realizar el agente situado según las decisiones tomadas por el razonador (el título de las acciones está definido por la ontología que utiliza el razonador en la comunicación para referirse a estas):

- **MoveToNode**: Como su nombre indica, esta acción consistirá en el desplazamiento del agente situado a un nuevo nodo. Adicionalmente, se hará una comprobación en la que se mirará que el nodo del que partimos es en el que estamos, de esta manera evitamos posibles movimientos ilegales que podrían provocar la muerte del agente. El mensaje contendrá el nodo del que se parte y al que se quiere ir.

- **SendAgentPositions**: El agente situado recibirá un mensaje que contendrá un *ACLMessage* con la información de las últimas posiciones de los agentes con los que se ha encontrado. Esta información se compartirá con el resto de agentes de la plataforma.

- **SendResourcesInformation**: Lo mismo que el caso anterior pero con la información de los recursos vistos por el agente.

- **SendMapTopology**: Realiza el mismo comportamiento que los dos casos anteriores pero compartiendo ahora la topología del mapa que el agente en cuestión conoce.

- **PickUp**: Como su nombre indica, esta acción consistirá en la recogida de minerales por parte del agente situado. Adicionalmente, se llamará a una función auxiliar que actualizará el *belief* que almacena el espacio libre de la mochila para que el agente BDI tenga en todo momento información verdadera. En este caso el contenido del mensaje es totalmente irrelevante.

- **OpenLock**: Como su nombre indica, esta acción consistirá en la apertura de una cerradura que se encuentre en la misma casilla que el agente situado. El contenido del mensaje indicará que tipo de mineral se encuentra detrás de la cerradura para poder así abrir con éxito el tesoro.

- **DepositIn**: Como su nombre indica, esta acción consistirá en la dejada de recursos en un agente almacenador que se encuentre contiguo a nosotros. Adicionalmente, se llamará a una función auxiliar para poder actualizar el espacio libre de la mochila del agente situado. El contenido del mensaje será el nombre del silo en el que se quieren almacenar los recursos.

Como hemos podido ver, el agente situado no es nada más que un enlace entre el razonador y el ecosistema que representa el mapa y el entorno de la partida. Vamos ahora a fijarnos en el comportamiento y toma de decisiones que realiza el agente BDI. Concretamente, vamos a comentar la función *treatObservationCollector* que será la encargada de razonar el comportamiento del agente recolector.

Lo primero que se hace es, si es necesario, resetear el *belief occupiedPositions* que define las posiciones ocupadas por agentes, como parece evidente, es necesario que cada poco tiempo reseteamos esta información ya que los agentes ya no se encontrarán ahí. A continuación comprobaremos si en nuestro nodo actual hay algún tesoro accesible o con posibilidad de ser accedido, y que obviamente sea de nuestro tipo. Si es así, se enviarán mensajes con las ontologías *PickUp* o *OpenLock* en función de si se quiere coger el tesoro o es necesario abrirlo previamente, además se enviará en el mensaje la información necesaria para la correcta ejecución de la acción.

Si no hay nada, se hará una comprobación de si la mochila está completamente llena o no. Si no nos queda espacio libre, buscaremos entre todos los agentes de los que tenemos información, cuáles de ellos son almacenadores que aceptan nuestro tipo de mineral. Entre todos estos, se buscará si alguno está entre las casillas ocupadas que conocemos, si encontramos un almacenador que sí, se trazará un camino hasta él mediante algoritmos de *dfs* y se enviará al agente situado un mensaje con ontología *moveTo* con el nodo al que ir. En caso de que no haya suerte, se buscará también si alguno de estos almacenadores los hemos visto alguna vez, cuya información entonces está almacenada en la estructura *agentPositions*, cabe destacar que esta información es menos fiable que la obtenida con *occupiedPositions* ya que la primera no se va restaurando, aunque si actualizando si nos los encontráramos otra vez. Si tampoco ha habido suerte, se hará un *dfs* al nodo no visitado más cercano que tengamos y el agente situado se moverá a ese nodo. Adicionalmente, cuando estemos en una casilla adyacente a un recolector, se enviará un mensaje con ontología *DepositIn* y el nombre del recolector al agente situado.

En el caso en el que la mochila no esté completamente llena, se buscará entre la información almacenada de los recursos del mapa, cuáles de ellos son de nuestro tipo y son accesibles, si alguno cumple con estas características, se irá hacía él. El funcionamiento es similar al que hemos visto con los almacenadores, se buscará un mineral al que queramos ir y, en el caso que sea posible trazar un camino hasta él, se enviará al agente situado un

mensaje con ontología *moveTo* y el nodo al que tiene que desplazarse, junto al nodo en el que estamos para la comprobación de que no se hagan movimientos ilegales.

Al final de esta función, se actualizará la base de datos de la información del mapa que tenemos con la observación recibida del agente situado para ir asegurando que la información que guardamos sea lo más verídica posible.

Destacar adicionalmente, que los agentes almacenadores del sistema irán enviando constantemente mensajes con la ontología definida como *TankingPositionInform*. Por tanto, el agente recolector tendrá un tratamiento especial para poder evaluar este tipo de mensajes cuando se redireccione al agente BDI. Sólo si la capacidad del recolector está por debajo del 80 por ciento, se evaluarán este tipo de mensajes, en cualquier otro caso, el mensaje será ignorado y el agente realizará sus estrategias ya explicadas. Cuando en cambio estamos por debajo del 80 por ciento de capacidad, sale rentable desviarse para depositar minerales, recordemos que si os ha llegado el mensaje es que el agente almacenador está relativamente cerca. Del mensaje en cuestión extraemos el nodo en el que se encuentra y aplicaremos sobre este un algoritmo para encontrar el mejor camino y dirigirnos hacia él. Una vez estemos adyacente a dicho agente, haremos el tratamiento habitual de depositar los minerales en el depósito del almacenador y después continuar con nuestras acciones.

Almacenamiento (tanker)

El principal objetivo de los agentes de almacenamiento es maximizar la cantidad de recursos transportados durante la ejecución de la partida. Los agentes de almacenaje puede almacenar recursos de uno o de ambos tipos de recursos, durante el registro del dicho agente le designaremos como “agentTankerDiamond” y/o “agentTankerGold”, para facilitar el almacenaje de cara los demás agentes de tipo recolector.

El agente situado actúa de distintas maneras a partir de las distintas ontologías del agente BDI. A continuación, vamos a comentar las distintas acciones que podrá realizar el agente situado según las decisiones tomadas por el razonador.

- **MoveTo:** esta acción consiste en desplazar el agente a un nodo en el que es adyacente nodo actual del agente situado, antes de eso comprobamos que el nodo que partimos es el nodo actual del agente situado y el nodo que queremos desplazar no es nulo, mediante el mensaje recibido que contiene el nodo de partida y nodo que se quiere desplazar.

- **SendAgentPositions**: este mensaje contiene la información sobre las últimas posiciones de los agentes con los que se ha podido encontrar. Y se compartirá con el resto de agentes de la plataforma.

- **SendResourcesInformation**: es un mensaje igual que el caso anterior, pero con la información sobre los recursos vistos qua ha podido ver el agente.

- **SendMapTopology**: comportamiento igual que los dos casos anteriores, pero compartiendo ahora la topología del mapa que el agente en cuestión conoce.

- **SendTankingPositionInform:** El agente situado recibirá un mensaje que contendrá un *ACLMessage* con la información de la última posición del agente situado y su tiempo. Esta información se compartirá con el resto de agentes de la plataforma.

Ahora vamos a fijarnos en el comportamiento y toma de decisiones del agente BDI de los agentes tankers, con la función *treatObsevationTanker* razona sobre el comportamiento del agente.

Inicialmente se comprueba que el belief *mapRepresentation* no está vacío, en caso afirmativo, se inicializa el mapa con el nodo situado del agente actual. Y si es necesario, resetear el *belief occupiedPositions* que define las posiciones ocupadas por agentes.

Se ha decidido que el agente tanker se mueva por el mapa haciendo movimientos que molesten lo menos posible a los otros agentes, especialmente a los agentes recolectores en su búsqueda de recursos.
Por este motivo, lo primero que se comprueba es si se encuentran en un nodo llamado *nodo a_salvo*, es decir, es un nodo que se caracteriza por tener una única adyacencia y no tiene

ningún tipo de recurso. Al encontrarse encima de este nodo, se asegura que el tanker no va a molestar a ningún otro agente, ya que no es de interés para ningún otro agente situarse en este nodo. Al situarse encima de este nodo, el tanker no hace más movimientos durante la partida y va enviando mensajes continuamente sobre su posición y con la ontología definida como *TankingPositionInform* para que los agentes recolectores sepan su localización y vengan a depositar sus recursos.

En caso de que el nodo actual no sea un *nodo a_salvo*, la misión del almacenador, como se ha mencionado anteriormente, es la de moverse de forma cuidadosa sin bloquear el paso a los recolectores. Es decir, mediante la utilización del belief *agentPositions* i *foreignAgents*, se comprueba si tiene agentes de tipo *"agentCollect"* i recursos, ya sea oro o diamante, que se encuentran cerca del propio agente, es decir, mediante un algoritmo de BFS se encuentra el camino más corto entre el agente de tipo *"agentCollect"* o los recursos con el propio agente, y si el camino es de tamaño menor que 4 es cuando se encuentran cerca. En caso de tener recolectores y recursos cerca, entonces mediante otro algoritmo de BFS se intenta predecir el camino más corto entre el agente de tipo *agentCollect"* y el recurso, este camino se almacena y el agente intenta evitar pisar esos nodos de la menor manera posible, y así facilitarles el paso hacia el recurso.

Si no se da el caso de tener agentes recolectores y recursos cerca de él, entonces el agente va en búsqueda de ese *nodo a_salvo* que pueda haber en el mapa. Es decir, del mapa que le puedan proporcionar los diferentes agentes de la partida, intenta ir a los que no ha visitado aún y tengan de momento una adyacencia.

Una vez hecho todo esto, puede darse el caso de que se tenga que mover del nodo actual o no, en caso de no tener que moverse envía un mensaje sobre su posición y con la ontología definida como *TankingPositionInform* para atraer a recolectores y que dejen sus recursos. Y en caso de que tenga que moverse, envía un mensaje sobre el nodo al que debe moverse con la ontología *moveTo*.

Finalmente, se actualizará la base de datos de la información del mapa que tenemos con la observación recibida del agente situado, es decir, de diferentes parámetros como la posición de los agentes y de los recursos y la topología del mapa, todo esto para ir asegurando que la información que guardamos sea lo más verídica posible.

En conclusión, se ha implementado el agente tanker más en función del objetivo colectivo, es decir, la de intentar recaudar todos los recursos del mapa, en lugar de dar énfasis sobre el objetivo individual. Ya que al tratar de evitar cruzarse con agentes recolectores cuando hay recursos cerca, se intenta "ayudarles" a poderlos recaudar. En caso de fomentar más el objetivo individual, se intentaría ir en búsqueda de recolectores todo el rato, pero esto

podría provocar el colapso y bloqueos a los recolectores y podría provocar la no recolección total de los recursos. La otra decisión importante es la de buscar los *nodos a_salvo*, esto se ha hecho también para intentar no bloquear a los recolectores y se ha decidido que al encontrar un nodo de estos mantenga durante toda la partida esa posición, ya que al tener una posición fija dentro del mapa y ir comunicando su posición, los recolectores lo tendrán más fácil para encontrarles que a otros tankers que se muevan continuamente.

Organización y distribución de las tareas

- Características principales del agente BDI y el agente situado: Jia Long Ji Qiu
- Agente explorador: Jia Long Ji Qiu
- Agente recolector: Victor Teixidó
- Agente almacenador Jiabo Wang, You Wu