Laboratorio 1

Desarrollo Basado en Agentes

Grupo 305 WING

UNIVERSIDAD DE GRANADA

I Lab 1: Patrolling Dagobah

1 Diseño del Agente

En este primer laboratorio hemos implementado un agente que representa un vehículo aéreo de tipo *Tie Shuttle*. Este agente está basado en los que creamos en los Casos Prácticos finales de tipo **STF**,

Para obtener el código inicial fusionamos los ficheros del github de tipo STF, en los que no habíamos profundizado demasiado en los casos prácticos, y obtuvimos un agente idéntico al *STF_DELIBERATIVE*.

A la hora de transformarlo y crear nuestro propio agente **Tie Shuttle** no incluimos grandes cambios en la forma de percibir el mundo ni en la forma de tomar decisiones, que ya sabemos que depende de la lectura de los sensores. Esto quiere decir que hemos mantenido la función de leer percepciones tal y como estaba, en la que se pide al Session Manager (SM) la información de los sensores y se le pasa a nuestro entorno, para que más adelante se puedan tomar las decisiones adecuadas respecto a la ruta a seguir.

Sin embargo, sí que hemos añadido cambios en la estructura de estados del agente, así como en la comunicación entre agentes, tal y como veremos a continuación.

Todos los diagramas se encuentran disponibles en el *siguiente enlace*, en caso de que sea necesario visualizarlos de una forma más clara.

2 Diagrama de estados

Como vemos, hemos mantenido la estructura inicial de estados, realizando cambios en los siguientes:

- **JOINSESSION**: tenemos dos pasos principales:
    - El primero se encarga de pedir las ciudades del mundo en el que estamos al Session Manager, da la opción al usuario de elegir la que quiera, y manda una solicitud de inicio de sesión en esa ciudad. Después de recibir la respuesta, inicia los NPC [1].
    - El segundo se encarga de solicitar la misión al usuario, que tendrá que ser la misma que la elegida al crear la sesión.
- **SOLVEPROBLEM**: en este estado iteramos a través de los objetivos de la misión (mientras ésta no esté completada), si el objetivo considerado es de tipo *LIST* o *REPORT* lo solucionamos convenientemente, y si es de tipo *MOVE IN*, pasamos al estado **SOLVEGOAL**.
- **SOLVEGOAL**: en este estado utilizamos el AUTONAV para ir estableciendo rutas a objetivos intermedios. Mientras no se esté en la ciudad objetivo, buscamos una ruta nueva, y tras eje-

[1] En nuestro caso, hemos probado con uno de tipo DEST y dos de tipo BB1F, pues ya de por sí a nuestros ordenadores les costaba finalizar las misiones en un tiempo razonable.

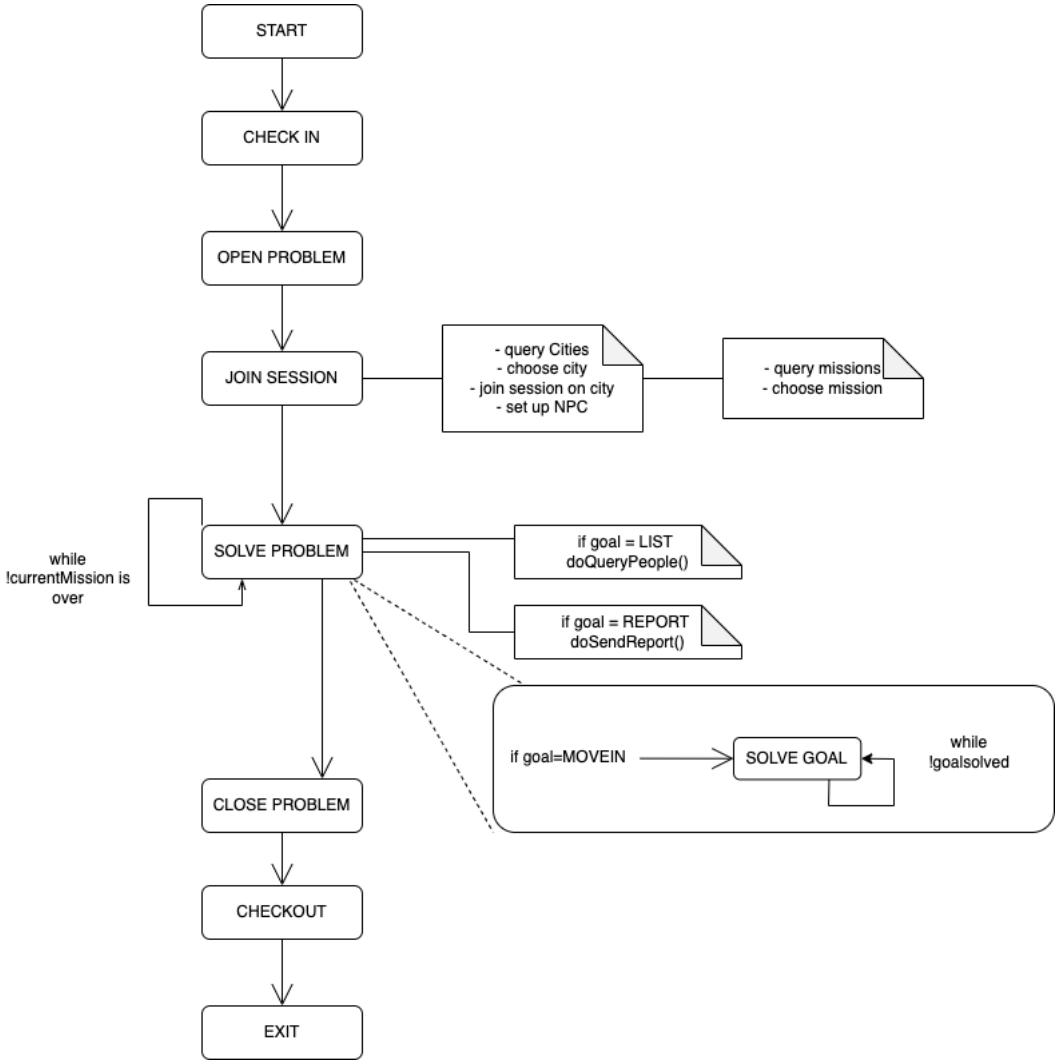
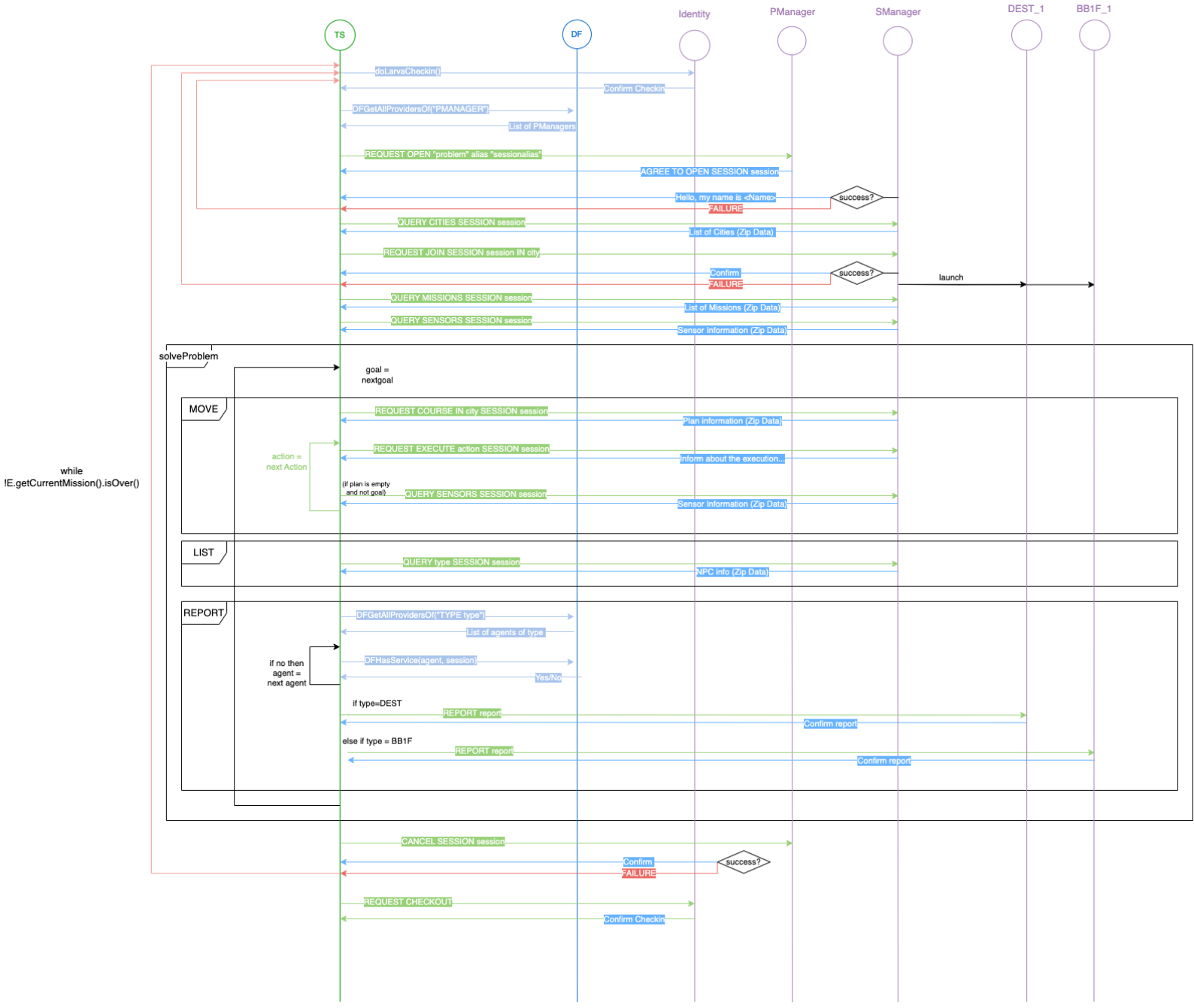


Figura 1: Diagrama de estados del agente TS

cutar todos los pasos, leemos los sensores, para que la siguiente búsqueda de ruta se haga correctamente. Una vez lleguemos a la ciudad objetivo, volvemos al estado **SOLVEPROBLEM**.

3 Diagrama de secuencia



En este diagrama también vemos los cambios descritos en el apartado anterior: antes de entrar en el bloque principal de SolveProblem, se mandan los mensajes de búsqueda de ciudades y de misiones al **SessionManager**. Una vez entramos en el bloque principal, volvemos a tener las tres opciones que ya comentábamos antes: LIST, REPORT y MOVE IN.

En la opción `LIST`, tenemos que mandar un mensaje al SM solicitando el número de personas de tipo conveniente. En la opción `REPORT`, primero tenemos que pedirle al DF todos los agentes que sean del tipo que queremos (DEST o BB1F) e ir verificando uno por uno cuál se encuentra en nuestra sesión. Después de comprobarlo, le mandamos el la información correctamente codificada. Por último, en la opción `MOVE IN` los mensajes para que el AUTONAV funcione se mantienen exactamente igual.

4 Diagrama de clases

Hemos decidido no incluir en la memoria el diagrama de clases, pues al tener un solo agente tenemos una estructura de clases clara, además resulta bastante ilegible. Sin embargo, en el diagrama hay cierta información sobre las funciones del agente que sí que nos gustaría explicar. En general las funciones son las que teníamos en los casos prácticos, aunque alguna tiene una estructura un poco distinta, como `myJoinSession()` o `mySolveProblem()`, tal y como hemos comentado en apartados anteriores. Sin embargo, hay otras funciones que hemos añadido:

- `mySolveGoal()`: es la que se encarga de iterar a través del plan generado por el AUTONAV hasta que éste se termine, entonces pasa a leer las percepciones y a generar un nuevo plan, mientras no se haya alcanzado la ciudad objetivo.
- `doQueryPeople()`: se encarga de solicitar al SM la información sobre el número de personas del tipo conveniente para cumplir el objetivo LIST. También almacena esa información en un diccionario local que servirá para mandar el report en caso necesario.
- `doSendReport()`: se encarga de preguntar al DF por los agentes del tipo que se solicite en el objetivo REPORT, elige a uno que se encuentre en su sesión, y le manda el report.
- `doCreateReport()`: se encarga de crear el mensaje con la codificación correcta, a partir de la información almacenada en el diccionario local mencionado con anterioridad, y vacía dicho diccionario.

5 Heurísticas utilizadas

En esta primera práctica hemos mantenido el **algoritmo A*** que nos proporcionaron los últimos casos prácticos.

Sin embargo, tras probar distintos problemas, nos dimos cuenta que no tenía sentido que el agente rodeara los obstáculos por un lado por defecto. Para solucionarlo, hemos introducido un cambio para que elija el lado por el que se rodea dependiendo de dónde está el objetivo.

Este cambio se da en la función `goAvoid()`, mantenemos el código inicial, pero separamos en dos casos: si el objetivo está a la derecha o si está a la izquierda, modificando la variable `nextWhichWall` según convenga. También modificamos la función `U()`, introduciendo la opción de seguir una pared por la derecha. Para esto último ha sido necesario crear las funciones que teníamos para seguir una pared por la izquierda, transformándolas para que cumplan la nueva funcionalidad.